APPENDIX I

Detailed Description And Figures to: U.S. Patent Application Serial No. 09/514,868 entitled "Apparatus and Method for Automated Selection of Markets and Routing to Markets of Orders for Securities" filed February 28, 2000, and *incorporated herein by reference.*

According to one embodiment of the invention, a system is provided for selecting markets for smart orders and routing smart orders to the selected markets. Figure 1 shows an overview of this embodiment of the invention. Customer workstations (102) are connected electronically through a data communications network (136) to order managers (105). The order managers (105) are connected electronically through data communications networks (138) to market services (108). Each operative market service (108) has associated with it a smart executor (116). The system of Figure 1, in some embodiments of the invention, implements multiple market services (108) and therefore multiple smart executors (116). Orders entered through customer workstations (102) are verified in order managers (105) and communicated across the networks (136, 138) to the market services (108) which pass the orders to the smart executors (116) for working storage in the order databases (118).

In the embodiment of Figure 1, each instantiation of a smart executor (116) has associated with it an order database (118). Although the order databases are referred to as "databases," a term that sometimes implies persistent disk storage, in fact, many embodiments of the invention will utilize order databases formed from STL container class objects maintained in high-speed, volatile memory such as random access memory or RAM, or other in-memory storage mechanisms.

A smart executor (116) also typically has access to a quote server library (124) of quotes kept current by a quote server (122) which obtains a constant stream of current quotes through data communications connections to markets (114) or to Nasdaq (126).

The smart executors (116) utilize the market information from the quote server library (124) to select a market (114) for an order. The smart executor then communicates to a market service (108) the identity of the market selected, and the market service (108) sends the order through a network (130) to the order router (112). The order router (112) sends the order across a network (132) to the order port (128) that services the particular market selected to receive the order. In the illustrated embodiment, one order port (128) is provided for each electronic market (114) to which orders can be sent.

The order ports (128) also receive from markets (114) responses to orders. Responses are communicated through networks (134, 132, 130, 138, 136) from the order ports (128) to the router (112), back to the market service (108), to the smart executors (116), which store in the order database (118) status information received in responses, back to the order managers (105), and back to the customer workstations (102) to provide to customers indications of order status.

The customer workstations, useful according to the present invention, comprise any configuration of computer workstation capable of inputting orders for securities and sending them in machine-readable form through a data communications network. The customer workstations, in some embodiments, day trader workstations in a broker/dealer office connected to a broker/dealer's internal network. In other embodiments, the workstations comprise home computers connected to an internet. In other embodiments, the workstations comprise office computers connected to an internet.

The data communications networks (130, 132, 134, 136, 138) in some embodiments comprise in-house networks within a broker/dealer's office. In other embodiments of the inventions, the data communications networks are internets, intranets, extranets, or any appropriate and useful data communications network arrangement. The networks (130, 132, 134, 136, 138) comprise any data communications networks capable of receiving and sending orders and notifications in electronic form among the customer workstations

(102), the order managers (105), the market services (108), the order router (112) the order ports (128), and the markets (114). In the embodiment of Figure 1, the networks (130, 132, 134, 136, 138) comprise separate data communications networks. In other embodiments, some or all of the networks comprise a single network.

Although they will appear in many embodiments, certain of the network connections are optional, depending on which embodiment of the invention is used. In certain embodiments of the invention, for example, the customer workstations (102) is directly connected to an order manager (105). Such direct connections are used, for example, in a broker/dealer office where day trading workstations are directly connected to the broker/dealer's order manager (105).

The order manager (105) comprises a computer capable of providing order management services in some embodiments. Examples of order management services include the usual known needs for administration of customer orders at the account level, including for example, customer logins, verifications of orders according to account codes, checking of customer credit eligibility for purchases and short orders, and regular business accounting. Order managers useful in accordance with the present invention are not required to provide each of these services, and, order managers will, in some embodiments, perform further services. The order manager (105) is any computer and software system capable of verifying orders and communicating them in a standard format to the market services (108).

It is noteworthy that the customer workstations, in some embodiments, belong to customers having accounts with the same broker/dealer running the market services 108), while, in other embodiments, they do not. The order managers (105) in some embodiments are run by the same broker/dealer running the market services (108). Other broker/dealers, however, in other embodiments, also, or even exclusively, make the market service available to their customers through a subscription arrangement with a

broker/dealer running market services (108) or smart executors (116). Customers (102) can gain access to the market service by opening accounts with a broker/dealer who runs market services on its own computers or by opening accounts with broker/dealers who subscribe to the market services run by others. From the perspective of the market services, it makes no difference whether orders originate from customers of the broker/dealer running the market services or from some other broker/dealers, so long as the orders arrive at the market services in a prescribed format intelligible to the market services. One embodiment of such an intelligible order format is shown in Figure 3, which is further described below.

Continuing with reference to Figure 1: The order manager (105) validates customer orders and then communicates them through data communications network (138) to a market service (108). In one embodiment, the market service (108) comprises a computer system, a combination of computing machinery and software, in which the software elements comprise a server which functions by continuously requesting, receiving from order managers (105), and sending to smart executors messages. Examples of the types of messages include orders and cancellations of orders. In some embodiments, messages include requests for other services such as symbol lists, refreshing pending order lists, or recovery services generally. Not all such message types are required, nor is this list of message types exhaustive.

In the illustrated embodiment, each market service (108) has associated with it a smart executor (116). The market service (108) starts its associated smart executor (116). While the smart executor is running, throughout the trading day, the market service (108) provides to the smart executor (116) an interface communicating orders, cancellations, and responses to and from the smart executor (116), the order managers (105), and the order router (112). In other embodiments, each market service has more than one smart executor, the number of smart executors being scaled according to a principle. In some embodiments the scaling principle is that one smart executor is assigned orders with

symbols beginning with letters in the range a..m, and a second smart executor is assigned orders with symbols beginning with letters in the range n..z. In other embodiments, other ordering principles are used.

The smart executor (116) comprises a software system installed and running on a computer, capable of maintaining an organized list of markets (114), receiving incoming orders (106) from the customer workstations (102) and the order managers (105), deciding automatically in response to programmed criteria which markets (114) ought to receive the orders, and sending the outgoing orders to the order router (112). The smart executor (116) is also capable of storing pending orders in an order database (118). Pending orders stored in the order database (118) have, for example, the structure shown in Figure 3, although alternative structures will occur to those of skill in the art without departing from the present invention..

The order router (112) comprises in some embodiments a computer system, a combination of computing machinery and software, capable of maintaining lists of routing connections, using the lists to select the best electronic route to a market, receiving from the market service outgoing orders in the form shown in Figure 3, and transmitting the outgoing orders through the selected route to a market (114).

The quote server (122), in the example illustrated in Figure 1, establishes and maintains electronic connections to markets (114) through which the quote server (122) receives quotes and other market information. The electronic connections to markets (114), in the illustrated example, include Nasdaq (126) feeds as well as direct connections to ECNs and other markets(114).

After orders are routed to markets, the order router (112) receives from the markets (114) responses, including confirmations of orders, notifications of executions of orders, and notifications of cancellations. Responses to order are generally communicated back

through the order router (112) to the market service (108), to the smart executor (116), to the order manager (105), and to the customer workstations (102).

Quotes from Nasdaq or Direct from Markets

As illustrated in Figure 1, orders for stock are originated on customer workstations (102), transmitted across electronic network connections (136, 138) and through the market service (108) to the smart executor (116). The market service (108) and its associated smart executor (116) are generally installed and running on at least one broker/dealer computer, although other configurations are within the scope of the present invention.

Figure 10 illustrated in more detail one embodiment of the smart executor of the present invention. In the embodiment illustrated in Figure 10, the smart executor (116) selects a market for receipt of an order by referring to a quote (1300) for the stock ordered. The quote (1300) is typically structured as shown in Figure 11. The fields or data elements comprising a quote are typically those shown in Figure 11, having the meanings described below. Quotes typically contain multiple sets or records comprising the fields Side (1320), Quantity (1322), MMID (1324), and Price (1326), which together represent offers to buy or sell the stock, the offers being displayed by markets, that is, ECNs or market makers. MMIDs (1324) or "market maker identifiers" are used as identity codes for ECNs as well as market makers.

Field	**Description**
Symbol (1302)	the symbol for the stock or other security whose market information comprises the quote.
LastTradePrice (1304)	the price of the stock in its last trade.
ClosePrice (1306)	the price of the stock in its last trade before closing in the previous trading day.

NetChange (1308)	the difference between LastTradePrice (1304) and ClosePrice (1306).
NationalMarket (1310)	the national market of the stock's last trade.
TypicalQuantity (1312)	the usual quantity in which the stock trades.
TickDirection (1314)	whether the current LastTradePrice (1304) is higher or lower than the LastTradePrice of the quote for the stock's most recent previous trade.
High (1316)	the stock's highest price in the current session.
Low (1318)	the stock's lowest price in the current trading session.
Side (1320)	whether a displayed offer is a bid or ask.
Quantity (1322)	the number of shares represented in a displayed offer.
MMID (1324)	the market symbol for the ECN or market maker displaying an offer.
Price (1326)	the share price in a displayed offer.
Cost (1328)	the transaction cost for purchases and sales of stock in the market identified by the corresponding MarketSymbol (1324).
IOC (1330)	a Boolean indication whether the market accepts IOC orders, yes or no, true or false.

In this embodiment of the invention, quotes comprise Nasdaq Level II Quotes. Other embodiments utilize quotes received through direct feeds from ECNs or other markets, or from a combination of Nasdaq Level II Quotes and direct feeds from ECNs or other markets.

As shown on Figure 10 for this embodiment, the smart executor identifies from the Price fields (1326) in the quote (1300) the markets, identified by the MMID fields (1324), that are quoting the inside price. In this embodiment, the smart executor then identifies, by use of the IOC fields (1330), which of the markets with the inside price will accept IOC orders. The process of identifying the markets with inside quotes willing to accept IOC orders is the process of generating the eligible market list (1202) as shown on Figure 10. In some embodiments, the smart executor gathers the markets quoting the inside price and willing to accept IOC orders, that is, the list of eligible markets (1206), into a data structure such as an array, linked list, or an STL container. Other embodiments use other means for implementing the list of eligible markets, all within the scope of the invention.

After generating the eligible market list (1202, 1206), the smart executor selects a market (1214) to receive the order. In this embodiment, if there is more than one eligible market, then the market having the lowest transaction cost is selected. If there is only one eligible market on the list (1206), then that market is selected. If there are no eligible markets, then this embodiment of the smart executor selects, for example, from among all IOC markets, the one having the lowest transaction cost. Other embodiments have other methods of selecting a market when no markets meet eligibility criteria.

If the quote (1300) changes after a market is selected, then this embodiment of the invention selects a new market. The market selection process (1214) detects changes while an order is pending by examining (1226) updated quotes (1224). Changes to be detected in the quote include, for example, changes in the inside price and changes in the number of markets quoting the inside price. If any such change occurs, the market is reselected by again generating the eligible market list (1202, 1206) and again selecting a market (1214) according to the content of the list.

After a market is selected, this embodiment of the smart executor (116) routes the order to the selected market by sending the order (1216) to the order router (112). During the

period when the order has been routed to a market but has not yet been executed, cancelled, rejected, or killed, the order is said to be "pending" or "floating." If the inside price improves while the order is pending, this embodiment of the smart executor responds by canceling (1218) the order. The cancellation process (1218) detects improvements in inside price by examining (1228) updated quotes (1224) from the quote server library (124).

Orders routed to markets are received (1230) by the markets (114). Markets receiving orders (114) transmit responses (1232) back through the router (112) to the smart executor (1234, 1220). Responses include confirmations of receipt and notifications of execution, cancellation, rejection, and killings.

Some embodiments of the smart executor include a control loop in which the steps of selecting a market according to the content of the signal (1214), routing the order to the selected market (1216), canceling the order if the inside price improves (1218), and receiving responses (1220) are repeated (1222, 1236) until the order is either filled, times out, or is cancelled, killed, or rejected.

Additional Aids of Execution Quality: Latency and Quantity of Securities Quoted

An alternative embodiment of the smart executor uses latency to select a market. According to this aspect of the smart executor, as shown on Figure 24, each market in a multiplicity of markets is assigned a latency (2404). Figure 24 is an example of an array in which latency (2404) is associated with markets identified by MMIDs (2402). Latency is a measure of the speed with which markets respond to orders. An example of a measure of latency is the difference between the time when a response is received from a market and the time when a corresponding order was routed to the market.

In this aspect of the invention, an array similar to the example in Figure 24 is the eligible market list (1206) generated (1202) by the smart executor, as shown on Figure 10. That is, the list of eligible markets (1206) includes the additional element of latency for each market. The step of selecting a market (1214) therefore is effected based on latency of the eligible markets. If the eligible market list (1206) identifies more than one eligible market, in one embodiment, for example, the step of selecting is carried out by selecting the eligible market having the lowest latency.

Alternative embodiments can combine latency with other parameters to select the market for the order. For example, as shown in Figure 25, in some embodiments markets identified by MMIDs (2402) are associated with transaction costs (2406) in addition to latency (2404). If the eligible market list contains more than one eligible market, the market to receive the order is, in these embodiments, selected by use of a combination of latency (2402) and transaction cost (2406), or by use of latency (2404) or transaction cost (2406) alone. For example, the step of selecting a market (1214) in some embodiments comprises selecting the market having the smallest product of latency multiplied by transaction cost.

Alternative embodiments can combine latency and transaction cost with other parameters to select the market for the order. For example, as shown in Figure 26, in some embodiments of the invention markets identified by MMIDs (2402) are associated with quantity of shares quoted (2408) in addition to transaction cost (2406) and latency (2404). If the eligible market list (1206 on Figure 10) contains more than one eligible market, in these embodiments, the market to receive the order is selected by use of a combination of latency, transaction cost, and quantity of shares. For example, in some embodiments, the selection comprises the market having the lowest quotient of quantity of shares offered for purchase or sale divided by transaction cost for execution of customer orders and further divided by latency.

Eligibility Not Limited To Inside Quote

In some embodiments of the invention, markets eligible for receiving orders are not limited to markets quoting the inside price. In such embodiments, the list of eligible markets is generated dependent upon one or more of latency, transaction cost, quantity of shares quoted, whether markets accept IOC orders, or other factors. In such embodiments, for example, the list of eligible markets is configured as an array that includes at least one of: latency, transaction cost, or quantity of shares quoted, as shown in figures 24, 25, and 26.

In some embodiments in which the list of eligible markets is not limited to markets quoting the inside price, the list of eligible markets is generated dependent upon coefficients whose values in turn depend upon some combination of latency, transaction cost, quantity of shares quoted, or other factors. The list of eligible markets in some embodiments, for example, is configured as an array that includes a coefficient whose values depends upon some combination of latency, transaction cost, quantity of shares quoted, or other factors. The list of eligible markets in such embodiments, is, for example, stored in a data container comprising an array, a linked list, a C-style data structure, a linked list of arrays, a linked list of data structures, an array of data structures, an STL template, a file, a database, or any other machine manipulable form.

Generating the eligible market list in this aspect of the invention optionally excludes from the eligible market list those markets having weighted coefficients above, equal to, or below a coefficient threshold, the coefficient threshold being calculated dependent upon some combination of quantity of securities quoted by the market, the market's transaction cost, and the market's latency. In some embodiments, generating the eligible market list includes ranking the markets in the list according to some ranking principle, such as, for example, latency or transaction cost, or a coefficient comprising the product of latency multiplied by transaction cost.

In the example embodiment illustrated in Figure 10A, selecting a market includes selecting (1240) a default market (1248). The default market (1248) in some embodiments is selected from among all available markets (1244). In other embodiments, the default markets is selected among eligible markets (1242). The default market (1248) in most embodiments is selected according to default market selection criteria (1246) which include, for example, transaction cost, latency, quantity of shares quoted, whether markets accept IOC orders, and other criteria.

In the example embodiment shown in Figure 10B, selecting a market includes configuring (1252) the order to comprise a limit order (1254) with time-in-force set to "IOC" and an order price set to an inside price, determining whether the order is marketable (1256), determining whether the order has been filled (1258), determining whether the order has been cancelled (1260), determining whether the order has been killed (1262), and determining whether the order has been rejected by all eligible markets (1264.

In the illustrated embodiment, the smart executor determines marketability (1256) by comparing the inside price with the order price. If the order price is better than the inside price on the quote, the order is not marketable. If the order price is equal to or worse than the inside price on the quote, the order is marketable.

In the embodiment of Figure 10B, if the order is marketable, not cancelled, not filled, and not rejected at the inside price by all eligible markets (1270), the method can include continuing to repeatedly select markets to receive the order by selecting the next eligible market (1268). In this aspect of the invention, upon the first routing of the order, 'the next market' is taken as the first eligible market in the eligible market list (1242). After routing the order to the last eligible market, 'the next market' is again the first eligible market.

In this embodiment of the invention, the smart executor will redetermine (1272) after each routing (1274) whether the order remains marketable (1256), not filled (1258), not cancelled (1260), not killed (1262), and not rejected at the inside price by all eligible markets (1264). In this embodiment, the smart executor continues to route the order to the next eligible market (1268, 1274) until market responses (1276) indicate that the order is either filled, cancelled, rejected at the inside price by all eligible markets, or the order becomes unmarketable.

If, in the embodiment shown in Figure 10B, the order is determined to be not marketable (1276), the order is then reconfigured (1278) to comprise the order's original price and the order's remaining time-in-force. Then order is routed (1280) to a default market (1248).

If, in the embodiment of Figure 10B, the order is determined to have been rejected at the inside price by all eligible markets (1282) and there exists at least one available markets not on the eligible market list (1288), the smart executor then selects (1284) an available market among the markets not on the eligible market list according to available-market selection criteria. The order is then routed to the selected available market (1286). Available-market selection criteria include, in this embodiment, for example, sending orders for national stocks to SOES and sending small-cap orders to SelectNet.

As shown on Figure 10B, the smart executor determines whether the order has been filled or cancelled by receiving (1276) from markets (114) responses to routed orders. The responses include, for example, notifications of execution, notifications of cancellation, notifications of rejections, and notifications of killings.

This aspect of the invention, as shown on Figure 10B, optionally includes the step of determining whether the inside price has changed (1296). If the inside price has changed,

this embodiment of the smart executor will reconfigure the order to the new inside price (1292), and then continue the determinations of marketability (1256), fill (1258), cancellation (1260), killing (1262), rejection (1264), selection of the next eligible market (1268), and so on.

Determining whether the inside price has changed (1296), in certain embodiments, includes marking the market where the order was pending when the inside price changed and continuing processing until the order is again routed to that market at the new inside price. Alternatively, other embodiments of the invention utilize a counter variable, count routings, reset the counter to zero when the inside price changes, and continue repeatedly determining marketability (1256), fill (1258), cancellation (1260), killing (1262), rejection (1264), selection of next market (1268), routing to next market (1276), and so on, until the counter value is equal to the number of eligible markets.

In this aspect of the smart executor, the core process of selecting markets to receive orders, in embodiments where the eligible list of markets is not limited to markets quoting the inside price, including the method of repeated selection of the next eligible market, functions generally as shown in the following example pseudocode:

```
Int SmartExecute.RequestCreateOrder(Order)
{
    // Order points to the subject order record
    (Order *) Order;

    // MarketList[] is an array of MMIDs representing eligible markets ranked
    // according to some ranking principle

    // Assume first entry in sorted list is
    // preferable default market ... although other criteria
```

```
// often will be used to select a default
int DefaultMkt = MarketList[1];

// Int MaxMkts = the current total number of eligible markets,
// i.e., the array size of MarketList[]
Int EndMKT = MaxMkts;          // set initially to max, so
                               // if the market does not change,
                               // then the loop will submit the
                               // order once to all IOC markets
Int TryMkt = 1;
Double TryPrice = CurrrentPrice = CIP(Symbol, Side)  /* Current Inside Price */

// continue to submit the order to eligible markets
// until the order is filled, cancelled, killed, or
// rejected by all eligible markets at the inside price
while(    Marketable(Order->Side, CIP, TryPrice) &&
          (Order->RemainingShares != 0)          &&
          (Order->CancelRequested != true)    &&
          (Order->TryMkt != MaxMkts))
{
    // index to the next eligible market or to the first eligible market
    if(++TryMkt > MaxMkts) TryMkt = 1;

    // submit order to current selected market
    OrderIDMkt = ifMarket->CreateOrder(MarketList[TryMkt], TryPrice, IOC)

    // wait for the entire order to be administered in the current markets,
    // i.e., entirely executed, cancelled, killed, or rejected
    while(Order->FloatingShares != 0)   /* blocking wait */   ;
```

```
            // if the inside price changes while order is pending in
            // the current market, mark the current market, so
            // that the loop will try all eligible markets at the
            // new inside price
            if((CurrentPrice = CIP(Symbol, Side)) != TryPrice)
            {
                EndMkt = TryMkt - 1;
                TryPrice = CurrentPrice;
            }
        } // end while( Marketable.....)

        // Order is now filled, cancelled, or unmarketable:
        // if order not filled and not cancelled, submit to default market
        // that is, if presently unmarketable, submit to default market
        if(!order_filled(OrderData) && !order_cancel_requested(OrderData)
            submit_order(DefaultMkt, OrderData.Price, OrderData.time-in-force);

    } // end RequestCreateOrder()


    Marketable(side, CurrentPrice, TryPrice)
    {
        if(side = buy)   return(TryPrice >= CurrentPrice);
        if(side = sell)  return(TryPrice <= CurrentPrice);
    }
```

Example Classes and Member Methods

In some embodiments of the invention, the core function and structure of the invention is implemented in three modules, a market service, a smart executor, and an order router. Class structures for object-oriented implementation of the market service and smart executor of such embodiments are illustrated in Figure 2. In such embodiments, the market service generally provides an interface between the smart executor (116) and the order managers (105) and the order router (112), as shown on Figure 1. The smart executor (116) performs the decision-making regarding where to route orders. The order router (112) administers the delivery of orders and cancellations to markets (114) and the receipt of responses from markets (114).

The market service (108) receives from order managers (105) and passes to the smart executor (116) by calls to ifSmartExecutorRequest.RequestCreateOrder(), a member method shown at reference number 212 on Figure 2, messages representing new orders in which the message data structures comprise the following fields, having the meanings and usages described, and illustrated on Figure 3. It should be noted that some embodiments of the invention utilize the same data structure, by altering the contents of pertinent fields, for both messages comprising orders and cancellations, as well as responses to orders to be communicated back from the market services to the order managers. Other embodiments utilize separate structures for different message types.

OrderIDMgr (302), a field containing a unique identifier for the order, assigned by the front-end, customer-level software that originated the order or by the order manager that sent the order to the receiving market service; this field is unique among orders sent by the originating order manager;

ManagerID (304), an identification code for the order manager originating the order; note that this field in combination with OrderIDMgr comprises a completely unique identifier for the order;

AccountID (306), the customer's account number;

Action (307), an indication of the processing to be performed in response to the order; this field identifies, for example, whether the message containing it is to be treated as an order, a cancellation, or a response;

Symbol (308), the trading symbol identifying the stock or other security to be administered in response to the message, to be sold, sold short, purchased, cancelled, or responded to;

Side (310), an indication whether an order is to buy, sell, or sell short.

Quantity (310), the original quantity of securities to be sold or purchased;

Minimum (312), related to Quantity, the minimum number of shares to execute, in which "0" as a code can indicate that any quantity is acceptable, and entry in this field of an amount equal to Quantity can mean that no partial fulfillment is allowed, that is, all of the Quantity is to be sold or purchase, or none of it.

Price (316), in which "0" can be treated as a code indicating a market order, and non-zero can indicate a limit order;

Time-in-force (318), sometimes abbreviated as "TIF," an integer indicating the number of seconds the order is to remain in force, can be coded so in the range 0 through 99999, "0" can be treated as indicating an IOC order;

MarketID (320), the market identifier code, generally an MMID, except when specifying SelectNet.

Parameters (322), for smart orders this field can contain the customer's price/speed preference; for orders directed to SelectNet, this field contains an MMID;

Sequence Number (324), a sequentially incremented number for transaction tracking between an order manager and a market service, this field helps guard against failure in the order manager, the market service, or the connecting network, after which failure both the market service and the order manager can exchange the most recent Sequence Number received by either and if there's a gap, fill it in.

Visibility (326), an indication whether the market to receive the order should display in a quote the order's price, symbol, and quantity; this field can be coded with the following values: HIDDEN means no display, SUBSCRIBER means display only to the market's direct subscribers, NORMAL means display according to the market's usual rules for display.

The smart executor (116) intelligently decides where to route orders, but the smart executor of the present invention does not verify orders. The smart executor assumes that orders it receives have already been verified. In describing the overall operation of the smart executor, we refer to order validation functions as residing in an order manager (105). Any order manager (105), however, can be used with the present invention so long as the order manager (105) is capable of (1) establishing a telecommunications link with the market service of the present invention by use of data communications procedures and (2) transmitting and receiving through the telecommunications link in established formats messages comprising orders, cancellations of orders, and responses regarding orders.

The smart executor, in embodiments of the kind illustrated on Figure 2, comprises three principal object-oriented classes, ifSmartExecutorRequest (202) and ifMarketEventSink (208), and SmartExecutor (210). In describing the kind of embodiment shown on Figure 2, the term "smart executor" is used to refer generally to the structure and function of these three classes. The small "if" on the front of several of the class names on Figure 2 denotes classes generally functioning as interfaces. The term "sink" which sometimes appears in class names denotes classes generally functioning to process responses or outputs. In the kind of embodiment illustrated on Figure 2, ifMarketEventSink (208) provides member functions generally oriented to administer responses from markets. The class IfSmartExecutorRequest (202) generally provides the input side of the interface to the smart executor.

Market services in the kind of embodiment shown on Figure 2 generally comprise three principal object-oriented classes, ifOrderEventSink (207), ifMarket (204), and ifExecutorEventSink (206). In describing such embodiments, the term "market service" is used to refer generally to the structure and function of these three classes. The class ifOrderEventSink (207) provides member methods generally designed to administer inputs from order managers, including requests for orders, cancellations, and recovery support. ifExecutorEventSink (206) provides member functions generally oriented to administer output from the smart executor. The class ifMarket (204) generally provides the interface between the smart executor and the markets. These embodiments of the invention generally will create and use multiple ifMarket class objects, one for each market to which orders may be directed. Although particular names are used for clarity in describing the structure and operation of the classes comprising such embodiments, obviously the names of object-oriented classes comprising embodiments of the invention have no effect on the structure, function, or scope of the invention.

In the illustrated embodiment, ifMarketEventSink (208) and ifSmartExecutorRequest (202) are abstract classes containing one or more pure virtual member methods required to be implemented by derived classes, if any. In the illustrated embodiment of the invention, only one class object will be derived from these abstract classes, the class called SmartExecutor (210). In this embodiment, a class called SmartExecutor (210) is derived from both ifMarketEventSink (208) and ifSmartExecutorRequest (202), and therefore there would be no class objects of ifMarketEventsink and ifSmartExecutorRequest in such implementations.

In the illustrated embodiment, class objects of ifExecutorEventSink (206) and ifMarket (204) are associated with the SmartExecutor (210) class object by aggregation, that is, by stored pointers. Member method names fully referenced would usually begin with the term "SmartExecutor," as in the example, "SmartExecutor.MethodNumberOne." In the

descriptions below, however, for clarity, member methods implemented in the derived class SmartExecutor (210) are named according to their base classes.

Because we use the term "smart executor" is used to refer generally to the three related interface classes shown on Figure 2, that is, the member methods of ifMarketEventsink (208) and ifSmartExecutorRequest (202) are referred to by their base class names, as in the examples, "ifMarketEventsink.MethodNumberOne()" and "ifSmartExecutorRequest.MethodNumberTwo()." In the illustrated embodiment, there is only one SmartExecutor class object for each market service. The illustrated embodiment of the invention, using only one market service, comprises only one SmartExecutor class object. Some embodiments of the invention use multiple instantiations of the SmartExecutor class in which each SmartExecutor class object is assigned responsibility for orders with symbols beginning with subsets of the subset of the alphabet administered by their associated market service.

General data flows among the principal classes of the market service and the smart executor are illustrated in Figure 2a. Order managers (105) generally send messages (294) requesting orders, cancellations, and services that are received by member methods in ifOrderEventSink (207), an interface of the market service (108). IfOrderEventSink (207) methods generally interface (295) with the smart executor (116) by calling member methods in ifSmartExecutorRequest (202). IfSmartExecutorRequest (202) in turn passes (296) orders and other requests to ifMarket (204) methods back in the market service (108). IfMarket (204) sends (297) market responses (297, 298) back through ifMarketEventSink (208) to ifExecutorEventSink (206) and then (299) back to the originating order managers (105).

In addition to the principal classes mentioned above, the illustrated embodiment, as shown on Figure 1A, also provides a container class called OrderDatabase (150) in which to store representations of orders. The invention provides representations of orders

through a class called Order (152). The OrderDatabase (150) typically comprises three STL containers (152, 154, 156), two of them (154, 156) used to encapsulate indexes into the third (152) which is used to hold data structures representing orders. In this embodiment, the first index (154) will sort according to the fields OrderIDMgr (191) and ManagerID (193), to allow quick access to orders according to the identification from the order manager side of the system. The second index (156) therefore will generally sort according to the field OrderIDMkt (192), allowing quick access to orders according to the identification of the order in the market to which the order is submitted.

The OrderDatabase, in the illustrated embodiment, as shown on Figure 1B, is a class providing the following member methods which function generally as described:

OrderDatabase.find(OrderIDMgr, ManagerID) (172) is a member method that returns an order given as an input the unique combination of order ID assigned by an order manager plus the identification code for the order manager (105) through which the order originated.

OrderDatabase.find(OrderIDMkt) (174) is a member method that returns an order given as an input the unique identification of the order in the market where the order is pending. In such embodiments, the member method OrderDatabase.find() is typically overloaded to function both with OrderIDMgr and managerID and also with OrderIDMkt .

OrderDatabase.newOrder(OrderMessage *) (176) is a member method that allocates space for a new order record.

OrderDatabase.indexOrder(OrderIDMgr, ManagerID) (178) is an overloaded member method for inserting the pertinent order fields into the first index.

OrderDatabase.indexOrder(OrderIDMkt) (180) is an overloaded member method for inserting the OrderIDMkt into the second index.

OrderDatabase.freeOrder(OrderIDMgr, ManagerID) (182) is a member method that deletes an order from the database given as an input the identification

of the order to be deleted. OrderDatabase.freeOrder() is typically overloaded (184) in such embodiments to function also with the OrderIDMkt field.

Order class objects represent orders internally within the smart executor. In the illustrated embodiment, as shown on Figure 1C, the Order class contains the member data elements described immediately below. One Order class object is instantiated for each order pending in the smart executor.

Order.Price (162) is the original limit price of the order or 0 for market order.

Order.TIF (164) is the order's time-in-force stated in seconds.

Order.Quantity (166) stores the original quantity of securities requested for sale or purchase.

Order.RemainingShares (168) is the quantity of securities, out of the original quantity stored in Order.quantity, remaining to be sold or purchased. If the order is subject to partial fulfillment, Order.remainingshares can be less than Order.quantity. When an order is fully executed, that is, "fulfilled," Order.remainingshares is set to zero.

Order.CancelRequested (186) is a Boolean flag set when the order is requested to be cancelled.

Order.FloatingMarket (188) is the identification of the market where the order is pending. In this context, "floatingmarket" is an MMID identifying the market where the order is pending.

Order.Parameters (190) is used for orders directed to SelectNet, typically will contain an MMID;

Order.OrderIDMkt (192) is the identification code of the order in the market where the order is pending.

Order.FloatingShares (194) is the number of shares on a floating order. If the order is subject to partial fulfillment, Order.floatingshares can be less than

Order.quantity. If the order is subject to partial fulfillment and is in fact partially executed, Order.FoatingShares will be less than Order.Quantity. After an order has been submitted to a market, the fact that the order is finished processing in that market is indicated by Order.FloatingShares == 0, because all shares on the order at that point have been either executed, cancelled, killed, or rejected.

Order.FloatingPrice (196) stores the price on a pending order.

Order.FloatingIOC (198) is a Boolean flag set to specify that the order was sent to market as an IOC order and therefore will not require a cancellation to be sent to the market.

Order.TryMarket (197) stores the index in the array OrderedMarkets of the market where the order is currently pending. In smart orders, this field is maintained by the smart executor. In normal orders, this field identifies the market selected by the customer.

Order.ScannedMarkets (195) stores the current count of the number of markets to which the order has already been submitted at the inside price.

Order.IsSmart (193) is a Boolean identifier of smart orders, orders for which automatic market selection has been requested.

Order.OrderIDMgr (191) is an identification code for an order, this code being assigned by the originating order manager or software operating on the customer workstation.

Order.ManagerID (193) is an identification code for the order manager that originated the order. ManagerID (193) and OrderIDMgr (191) together comprise a unique key for an order in the order database.

The list of eligible markets (MMIDs, reference 352), in some embodiments, is stored in an array called OrderedMarkets (350), as shown on Figure 1D. Some embodiments of the invention also allocate an array dimension for weighted coefficients (354). In some embodiments, within OrderedMarkets (350), markets represented by MMIDs (352) are

sorted according to the weighted coefficients (354). Weighted coefficients, in some embodiments, are calculated with some combination of market transaction costs, market latency, or quantity of shares quoted.

The SmartExecutor Class (reference 210 on Figure 2)

As mentioned above, in this embodiment of the invention, the SmartExecutor class (210) implements through inheritance the classes ifSmartExecutorRequest (202) and ifMarketEventSink (208), shown on Figure 2. In addition to the member methods defined in those classes, the SmartExecutor class (210) generally also implements the three private functions described below.

SmartExecutor.smartNewOrder(Order *) (reference 216 on Figure 2)

In the illustrated embodiment, smartNewOrder() (216) is called from ifSmartExecutorRequest.RequestCreateOrder() (212)to initialize the order fields for initial processing and pass the order to smartContinueOrder() (220), as shown on Figure 9. (The order is actually created in ifSmartExecutorRequest.RequestCreateOrder() (212).) More specifically, in the illustrated embodiment, smartNewOrder() (216) functions in accordance with the following pseudocode:

```
SmartExecutor.smartNewOrder(Order *)
{
    //initialize order fields for smart processing
    Order->TryMkt = 0;
    Order->ScannedMkts = 0;
    Order->floatingPrice = CIP(Symbol, Side);  // current inside price
    smartContinueOrder(Order);
```

}

SmartExecutor.smartCancelOrder(Order *) (reference 218 on Figure 2)

In the illustrated embodiment, this method marks orders as having been requested to be cancelled. More specifically, smartCancelOrder() functions in accordance with the following pseudocode:

```
SmartExecutor.smartCancelOrder(Order *)
{
        Order->cancelRequested = true;
        // remember no need to cancel a pending IOC order
        If(not Order->floatingIOC)
                IfMarket->CancelOrder(Order);
}
```

SmartExecutor.smartContinueOrder((Order*) Order) (reference 220 on Figure 2)

In the illustrated embodiment, this method decides which market is to receive a smart order. This method is called on every response from markets regarding smart orders and from SmartExecutor.newOrder(), reference 220 on Figure 9. This method is not called if there are no remaining shares to be executed on the order, that is, if Order->RemainingShares == 0. See the description of ifSmartExecutorRequest.NotifyOrderExecuted().

The following example implementation can be compared with the one set forth above for SmartExecute.RequestCreateOrder(Order). In comparison, this implementation uses no blocking waits while orders are pending, because, in this embodiment, the call to sendToMarket() represents a handoff to a background or parallel process. In high-volume environments, blocking waits may actually be preferred, especially if a handoffs to parallel processes requires context switches. Nevertheless, many embodiments within the scope of the invention do not use blocking waits. More specifically, smartContinueOrder(), in this example embodiment, functions in accordance with the following pseudocode:

```
SmartExecutor.smartContinueOrder((Order *) Order)
{
        // first, wait until the pending order is
        // finished processing in the current market,
        // i.e., until fully executed or rejected
        if(Order->floatingshares != 0)  return;

        if(Order->CancelRequested)
        {
                ifExecutorEventSink->NotifyOrderCancelled(Order);
                OrderDatabase.freeOrder(Order);
                Return;
        }

        if(timed_out(Order))
        {
                NotifyUserRejected(timeout_as_reason);
                OrderDatabase.freeOrder(Order)}
```

```
        Return;
    }

    if(marketable(Order))
    {
        if(Order->FloatingPrice != CIP(Symbol, Side))
        {
            //price changed will need to restart count
            Order.FloatingPrice = CIP(Symbol, Side);
            Order->ScannedMarkets = 0;
        }
        if(Order->ScannedMarkets < arraysize(OrderedMarkets))
            //send to current market with IOC as TIF and
            //floating price as limit price
            //increment market cycle back to beginning:
            sendToMarket (Order, IOC,
                    OrderedMarkets[Order->TryMarket]);
            // point to next market
            Order->TryMarket += 1 mod array_size;

        else    // finished scan of all markets on eligible list,
            // still marketable,
            // mark as not smart anymore, normal order,
            // and send to SOES or SelectNet
        {
            Order->FloatingPrice – Order->Price;
            Quote* quote – QuoteServer.getQuote(Order->Symbol);
            If(quote->NationalMarket == "N")
```

```
                                sendToMarket(Order, remainingTIF(Order.TIF),
                                "SOES");
                        else    // if small cap
                                {
                                Order->Parameters = "SNET_PREF = "
                                        + quote.topMMID;
                                SendToMarket(Order, remainingTIF(Order.TIF),
                                        "SNET")
                        } // end elseif small cap
                } // end if (Order->scannedMarkets .....)
        } // end if marketable
        else // non marketable
        {
                // send to default market with remaining timeout
                // and at the original order price
                // and marked as a normal, non-smart order
                Order->FloatingPrice = Order->Price;
                sendToMarket(Order, remainingTIF(Order.TIF), DefaultMarket);
        } // end else

} // end smartContinueOrder()

sendToMarket((Order *) Order, RemainingTIF, MarketPtr)
{
        Order->floatingShares = Order->RemainingShares;
        Order->TIF = RemainingTIF;
        MarketPtr = ifMarket->CreateOrder(Order);
}
```

The ifOrderEventSink Class (reference 207 on Figure 2)

The ifOrderEventSink class provides the member methods for interface between the order managers and the smart executor. The member methods of ifOrderEventSink, described below and identified on Figure 2, are Startup() (290), ReceiveOrderMessage() (292), and Shutdown() (293).

Startup() (reference 290 on Figure 2)

In the illustrated embodiment, ifOrderEventSink.Startup() is called from the operating system as an executable to begin operation of the entire market service and smart executor. At startup time, as shown on Figure 5, a market service (108), that is, the member method ifOrderEventsink.Startup() (290), first can call (512, 516, 520, 524, 528, 532) constructors (502, 504, 506, 508, 510) for ifMarketEventSink , ifSmartExecutorRequest, SmartExecutor, ifExecutorEventSink, and ifMarket.

In the illustrated embodiment, only one class object is instantiated for ifMarketEventSink, ifSmartExecutorRequest, SmartExecutor, and ifExecutorEventsink. In embodiments in which the class SmartExecutor is implemented by inheritance from ifMarketEventSink and ifSmartExecutorRequest, then the market service starts up by calling only the constructor for SmartExecutor instead of the constructors for ifMarketEventSink and ifSmartExecutorRequest. In typical embodiments, one class object for ifExecutorEventsink is created.

In the illustrated embodiment, multiple specializations or derived instances of ifMarket are constructed (510), one for each market that can receive orders. The specializations of ifMarket are derived class objects, one for each market, each

containing private member methods designed to work with a particular market. More specifically, the specializations of ifMarket each implement the requirements of the well-known communications protocol of a particular market.

In the illustrated embodiment, the constructors (502, 504, 506, 508, 510) all return pointers to the class objects constructed (514, 518, 522, 526, 530, 534) for storage in the market service (108). The constructor call (516) to ifSmartExecutorRequest() passes as a parameter for storage in ifSmartExecutorRequest the pointer to the ifMarketEventSink class object.

After constructing the class objects for the smart executor, in the embodiment illustrated, ifOrderEventSink.Startup() (290) makes multiple calls (602) to ifSmartExecutorRequest->RegisterMarket() (222) as shown on Figure 6, one such call for each ifMarket class object constructed.

Finally, ifOrderEventSink.Startup() uses known means to open a socket for data communications to be dedicated to messages to and from order managers. Startup() then registers ifOrderEventSink.ReceiveOrderMeassage() as the event-driven call-back function for messages received on the socket.

ReceiveOrderMessage() (reference 292 on Figure 2)

In the illustrated embodiment, this method is registered by ifOrderEventSink.Startup() with the data communications API as the call-back for messages received on a socket dedicated to handling messages to and from order managers. IfOrderEventSink.ReceiveOrderMessage() is always available to receive messages from order managers. IfOrderEventSink.ReceiveOrderMessage() functions by calling member methods in ifSmartExecutorRequest. IfOrderEventSink.ReceiveOrderMessage()

determines according to the Action code in messages which member method to call in ifSmartExecutorRequest.

That is, if the Action code indicates that a message from an order manager represents a new order, ifOrderEventSink.ReceiveOrderMessage() calls ifSmartExecutorRequest.RequestCreateOrder(). If the Action code indicates that a message from an order manager represents a cancellation, ifOrderEventSink.ReceiveOrderMessage() calls ifSmartExecutorRequest.RequestCancelOrder().

Shutdown() (reference 293 on Figure 2)

In the illustrated embodiment, this method is the endpoint for the entire system of the market service and smart executor. This method, like ifOrderEventSink.Startup() (290), is called from the operating system from an executable to begin operation of the entire market service and smart executor. This method functions, as shown on Figure 8, by calling (802) ifSmartExecutorRequest.shutdown() (226).

The ifSmartExecutorRequest Class (reference 202 on Figure 2)

The principal inputs to the smart executor are the member methods of ifSmartExecutorRequest which function as described below. As shown on Figure 9, the smart executor functionality in the illustrated embodiment is generally invoked by a call from the market service, that is, from ifOrderEventSink.ReceiveOrderMessage() (292), to the member method ifSmartExecutorRequest.RequestCreateOrder() (212) or to RequestCancelOrder() (214).

RequestCreateOrder (OrderMessage *) (reference 212 on Figure 2)

In the illustrated embodiment, this method is called by ifOrderEventSink.ReceiveOrderMessage() (292) when a create order request message is received from the order manager (105). This method reserves a record for the new order in the order database, sends normal orders to the market identified in the order, and sends smart orders to SmartExecutor.smartNewOrder(). More specifically, RequestCreateOrder() functions as shown in the following pseudocode:

```
RequestCreateOrder (OrderMessage *)
{
    (Order *) Order = OrderDatabase.newOrder(OrderMessage *);
    If(is_smart(Order))  smartNewOrder();
    else  sendToMarket(Order, TIF, Order->TryMarket);
}
```

RequestCancelOrder (CancelMessage *) (reference 214 on Figure 2)

Called by ifOrderEventSink.ReceiveOrderMessage() (292), this method is used in the illustrated embodiment to cancel an order previously placed into the smart executor. More specifically, RequestCancelOrder() functions as shown in the following pseudocode:

```
RequestCancelOrder (OrderID, ManagerID)
{
    Order* Order = OrderDatabase.find(OrderID, ManagerId);
    If(is_smart(Order))  smartCancelOrder(Order);
```

```
                else
                {
                        ifMarket* MarketPtr;
                        // load MarketPtr with a pointer to the market
                        // where the order is pending
                        MarketPtr = Markets(Order->FloatingMarket);
                        // and then cancel the order in the market
                        MarketPtr->CancelOrder(Order->OrderIDMkt);
                }
        }
```

RegisterMarket (Market *) (reference 222 on Figure 2)

In the illustrated embodiment, the market service startup function ifOrderEventSink.Startup() (290), at startup time, constructs multiple specializations of ifMarket, one for each available market, and retains in storage pointers to those ifMarket class objects. ifOrderEventSink.Startup() then calls RegisterMarket() once for each market, passing as a parameter a pointer to each ifMarket class object in turn.

RegisterMarket(ifMarket*) (222) functions, as shown on Figure 6, by calling ifMarket->MarketName() (234) which returns the name or MMID of the market served by the called specialization of ifMarket. RegisterMarket() (222) then stores the market name and the pointer in a lookup container such as a table, arrray, or linked list. Finally, RegisterMarket() calls ifMarket->RegisterMarketEventSink(pMarketEvenSink) passing a pointer to the ifMarketEventSink object so that each ifMarket object can direct calls to member methods within the ifMarketEventSink class object.

Startup () (reference 224 on Figure 2)

In the illustrated embodiment, this method is called from ifOrderEventSink.Startup() (290) to initialize the smart executor. Called after RegisterMarket() so that the smart executor now possesses pointers to the ifMarket class objects, ifSmartExecutorRequest.Startup(), illustrated on Figure 6, functions as follows.

First, ifSmartExecutorRequest.Startup() calls (904) ifMarket->Startup() once for each existing ifMarket class object.

Second, ifSmartExecutorRequest.Startup() calls (906) ifExecutorEventSink.NotifyStartup() to notify the order manager (105) that the smart executor is operative, ready to receive orders.

RegisterExecutorEventSink (ifExecutorEventSink *) (reference 228 on Figure 2)

As shown on Figure 7, this method (228) is called (916) by ifOrderEventsink.Startup() (290) at startup time to pass to the ifSmartExecutorRequest class object a pointer to the ifExecutorEventSink class object so that member methods in ifSmartExecutorRequest can call member methods in ifExecutorEventSink.

Shutdown () (reference 226 on Figure 2)

In the illustrated embodiment, ifSmartExecutorRequest.Shutdown() (226) is called by ifOrderEventSink.Shutdown() (293), as shown on Figure 8. ifSmartExecutorRequest.Shutdown() will cleanly shut down the smart executor by reading through the pending orders (1006) in the order database (118),

canceling (1008) all pending orders, and rejecting (1002) all new orders that might arrive during shutdown processing. If markets fail to respond to cancellations within some set period of time, shutdown processing will continue regardless. ifSmartExecutorRequest.Shutdown() continues shutdown processing by calling (1018) ifMarket->Shutdown(), one call for each specialization of ifMarket. After ifSmartExecutorRequest.Shutdown() has been called, ifSmartExecutorRequest.Startup() must be called before any additional orders will be processed by this implementation of the smart executor.

The ifMarket Class (reference 204 on Figure 2)

Derived specializations of ifMarket class objects provide smart executor interfaces to markets. By use of ifMarket class objects, all class objects derived from ifMarket, one for each available market, present the same interface to the smart executor. The ifMarket member methods function as described below. There will exist generally one specialization class object of ifMarket for each market capable of receiving orders from the smart executor.

CreateOrder (OrderData *) (reference 230 on Figure 2)

In the illustrated embodiment, this method is called by ifSmartExecutorRequest.RequestCreateOrder() (212) to enter an order in a market. This method assigns a completely unique identification code OrderIDMkt (192 on Figure 1C). When orders are received by the smart executor, they are uniquely identified by the combination of OrderID and ManagerID (191, 189 on Figure 1C). The identification code on the market side, however, must comply with the individual market protocols, be completely unique across all orders in the order database, and fit into one field. Therefore,

CreateOrder() assigns a new unique code to each order, uses the new code in the order message sent through the router to the market, and stores the code in the field Order->OrderIDMkt (192).

After assigning the new identification code, this method CreateOrder() (230), as shown on Figure 9, then converts the order data into the form required by the specific market protocol for the market to which the order is directed and transmits the order in the form of a message through a data communications connection, such as, for example, a tcp/ip connection, to the order router (112).

CancelOrder (pOrderData *) (reference 232 on Figure 2)

In the illustrated embodiment, this method is used to cancel an order pending in a market. This method is called by ifSmartExecutorRequest.RequestCancelOrder() (214) to cancel an order in a market. This method functions by sending to the router a cancellation message with two pertinent elements, the OrderIDMkt code (192 on Figure 1C) identifying the order to cancel and an Action code set to "Cancel." The cancellation is sent in the form of a message through a data communications connection, such as, for example, a tcp/ip connection, to the order router (112).

MarketName () (reference 234 on Figure 2)

In the illustrated embodiment, this method is called by ifSmartExecutorRequest.RegisterMarket() (222), as shown on Figure 6, to obtain the market name for the market administered by a ifMarket class object specialization. The market name is a character string such as "SNET" or "BTRD." There is one specialization object of the ifMarket class assigned to each market. This method marketName() (234) returns a string containing or pointing to the

market name for the market.

RegisterMarketEventSink (ifMarketEventSink *) (reference 236 on Figure 2)

In the illustrated embodiment, ifSmartExecutorRequest.RegisterMarket() (610) calls ifMarket->RegisterMarketEventSink(ifMarketEventSink*) (236), as shown on Figure 6, passing a pointer (612) to the ifMarketEventSink class object so that ifMarket class specializations can store the pointer and use it to direct calls to member methods within the ifMarketEventSink class object (reference 208 on Figure 2). In this embodiment, the only function of ifMarket->RegisterMarketEventSink(ifMarketEventSink*) (236) is to receive and store the pointer (612) to the class object ifMarketEventSink (208 on Figure 2) passed to it as a parameter from ifSmartExecutorRequest.RegisterMarket() (222).

Startup () (reference 238 on Figure 2)

Called by ifSmartExecutorRequest.Startup() (224) to establish a data communications connection to the order router (112), in the illustrated embodiment, this method ifMarket.Startup() (238) initializes the router connection. In some embodiments, after establishing the router connection, this method ifMarket.Startup() (238) calls ifMarketEventSink.NotifyMarketConnected() (288) to notify the smart executor that its associated market is available for orders. In some embodiments, ifMarket.Startup() calls ReceiveMarketMessage() to establish either an event-driven receipt of a market response, or a blocking receive for a market response.

In some embodiments, in the process of establishing the data communications link with the router, ifMarket->Startup() determines and stores in a member data element the data communications parameters needed by all the member functions

of ifMarket specializations for them to communicate with the router. If, for example, the data communications technology in use is tcp/ip, then ifMarket->Startup() will retain in storage the pertinent socket identification for the router.

ReceiveMarketMessage() (reference 233 on Figure 2)

In the illustrated embodiment, this method is called by ifMarket.Startup() at startup time and thereafter runs generally continuously, in those embodiments using blocking waits, using the previously stored data communications parameters, by repeatedly requesting to receive and subsequently receiving data communications packets (914) from the order router (112), as shown on Figure 7.

In embodiments designed for event-driven function, ReceiveMarketMessage() is registered by ifMarket.Startup() with the data communications API or middleware as a callback function for a router socket. In such embodiments, the API or middleware calls ReceiveMarketMessage() when receiving responses back through the order router from markets. Packets returning from the router to be received by ReceiveMarketMessage() represent responses to orders and cancellations.

The data communications API or middleware, in many embodiments, can call ReceiveMarketMessage() when the data communications connections to the router fails for any reason. That is, for example, a call to receive(socketID) returns an ERROR code. In blocking embodiments, the call from ReceiveMarketMessage() to the API will return an error code. In event-driven embodiments, the API or middleware can support a callback to ReceiveMarketMessage() passing an error message. In most embodiments, however, ReceiveMarketMessage(), upon being informed of a failure of its data

communications connection to the router, calls ifMarketEventsink.NotifyMarketDisconnected(ifMarket* this) (288 on Figure 2) to alert the smart executor that the associated market is not available for orders and then calls ifMarket->Startup() to reestablish the market service connection to the router.

Shutdown () (reference 240 on Figure 2)

In the illustrated embodiment, this method is called (1018) by ifSmartExecutorRequest.Shutdown() (226), as shown on Figure 8. This method functions by simply closing its data communications connection to the order router. If, for example, the data communications are effected through winsock or Berkeley sockets, this method simply closes the pertinent socket. This method then calls ifMarketEventSink.NotifyMarketDisconnected(ifMarket* this) (288).

The ifMarketEventSink Class (reference 208 on Figure 2)

The ifMarketEventSink class provides member methods generally oriented to administer responses from markets. The class member methods in the illustrated embodiment function generally as described below.

NotifyOrderConfirmed (OrderConfirmedData *) (reference 274 on Figure 2)

In the illustrated embodiment, this method is called when a market confirms an order, as shown on Figure 9. This method is called by ifMarket.ReceiveMarketMessage() (233) and in turn calls smartContinueOrder() or ifExecutorEventSink.NotifyOrderConfirmed(). More specifically, this method functions as shown in the following pseudocode:

```
NotifyOrderConfirmed (OrderConfirmedData*)
{
        Order* Order =
                OrderDatabase.find(OrderConfirmedData->OrderIDMkt);
        if(Order->IsSmart)  smartContinueOrder(Order);
        else  ifExecutorEventSink->NotifyOrderConfirmed(Order);
}
```

NotifyOrderRejected (OrderRejectedData *) (reference 276 on Figure 2)

In the illustrated embodiment, this method is called when a market rejects an order. This method is called by IfMarket.ReceiveMarketMessage() (233) and in turn calls ifMarketEventSink.NotifyOrderRejected(). This method functions as shown in the following pseudocode:

```
NotifyOrderRejected(OrderRejectedData *)
{
        Order* Order =
                OrderDatabase.find(OrderRejectedData->OrderIDMkt);
        Order->FloatingShare -= OrderRejectedData->RejectedShares;
        if(Order->IsSmart)  smartContinueOrder(Order);
        else  ifExecutorEventSink->NotifyOrderRejected(Order);
}
```

NotifyOrderExecuted (OrderExecutedData *) (reference 278 on Figure 2)

In the illustrated embodiment, this method is called when the market executes an order, as shown on Figure 9. This method is called by

IfMarket.ReceiveMarketMessage() (233) and in turn calls ifMarketEventSink.NotifyOrderExecuted(). This method in the embodiment illustrated compares the number of shares ordered with the number of shares executed, and, if there are no shares remaining for execution, the method releases the order record from the order database. In the illustrated embodiment, this method functions generally as shown in the following pseudocode:

```
NotifyOrderExecuted(OrderExecutedData*)
{
    Order* Order =
        OrderDatabase.find(OrderExecutedData->OrderIDMKT);
    IfExecutorEventSink->NotifyOrderExecuted(Order);
    Order->RemainingShares -=
        OrderExecutedData->ExecutedShares;
    if(Order->RemainingShares == 0)
        OrderDatabase.freeOrder(Order);
    else if(Order->IsSmart)
    {
        Order->FloatingShares -=
            OrderExecutedData->ExecutedShares;
        smartContinueOrder(Order);
    }
}
```

NotifyExecutionReport (ExecutionReportData *) (reference 280 on Figure 2)

In the illustrated embodiment, this method is called as a supplement to NotifyOrderExecuted() when a market executes an order. The purpose is to provide additional information to the order manager (105). The additional

information is the execution price and the counterparty. This additional step is needed because the additional information is generally not provided by the markets immediately upon execution, but is often provided later. Markets can take many seconds, a long time to wait when a stock is active, to provide price and counterparty. The execution notification is provided in two stages, therefore, so that customers are not required to wait for the second response to learn of their executions. IfMarket.ReceiveMarketMessage() calls ifMarketEventSink.NotifyExecutionReport(), which in turn calls ifExecutorEventSink.NotifyExecutionReport(), which creates and sends a data communications message to the originating order manager. NotifyExecutionReport() would not generally call smartContinueOrder() because NotifyOrderExecuted() already made that call.

NotifyOrderCancelled (OrderCancelledData *) (reference 282 on Figure 2)

In the illustrated embodiment, this method is called when the market cancels an order. This method is called by IfMarket.ReceiveMarketMessage() (233) and in turn calls smartContinueOrder() (220) or ifMarketEventSink.NotifyOrderCancelled() (282). This method functions as shown in the following pseudocode:

```
NotifyOrderCancelled(OrderCancelledData *)
{
    Order* Order =
        OrderDatabase.find(OrderCancelledData->OrderIDMkt);
    Order->FloatingShares -= OrderCancelledData->CancelledShares;
    if(Order->IsSmart)  smartContinueOrder(Order);
    else  ifExecutorEventsink->NotifyOrderCancelled(Order);
}
```

NotifyOrderKilled (OrderKilledData *) (reference 284 on Figure 2)

In the illustrated embodiment, this method is called when the market terminates an order before the order is executed or cancelled. This method is called by IfMarket.ReceiveMarketMessage() (233) and in turn calls smartContinueOrder() (220) or ifMarketEventSink.NotifyOrderKilled() 284). This method in the illustrated embodiment functions generally as shown in the following pseudocode:

```
NotifyOrderKilled(OrderKilledData*)
{
        Order* Order =
                OrderDatabase.find(OrderKilledData->OrderIDMkt);
        Order->FloatingShares -= OrderKilledData->KilledShares;
        if(Order->IsSmart  smartContinueOrder(Order);
        else   ifExecutorEventSink->NotifyOrderKilled(Order);
}
```

NotifyMarketConnected (MarketConnectedData*) (reference 286 on Figure 2)

In the illustrated embodiment, this method is called by ifMarket.Startup() when a specialization of ifMarket is connected for data communications to the order router. NotifyMarketConnected() (286) functions by setting to TRUE a member Boolean field, called for example "MarketConnected."

NotifyMarketDisconnected () (reference 288 on Figure 2)

In the illustrated embodiment, this method is called by

ifMarket->ReceiveMarketMessage() when the data communications connection to the order router is lost. If, for example, the data communications connection to the router is effected through a winsock socket, this method is called by ifMarket->ReceiveMarketMessage() (233) when the socket is closed unexpectedly. In some embodiments, this method is also called by ifMarket->Shutdown() when ifMarket->Shutdown() intentionally closes the data communications connection with the router. NotifyMarketDisconnected() functions in the illustrated embodiment by resetting to FALSE a member Boolean field, called for example "MarketConnected".

The ifExecutorEventSink Class (reference 206 on Figure 2)

The ifExecutorEventSink class provides member methods generally oriented to administer output from the smart executor. The class member methods function generally as described below.

NotifyOrderConfirmed (NotifyOrderConfirmedData *) (reference 242 on Figure 2)

In the illustrated embodiment, this method is called from ifMarketEventSink.NotifyOrderConfirmed() (274) when an order has been confirmed by a market. The market gave notification by sending a message back through the order router eventually received by ifMarket.ReceiveMarketMesssage() which then called ifMarketEventSink.NotifyOrderConfirmed(). IfExecutorEventSink.NotifyOrderConfirmed() (242) operates by sending through a data communications network a message notifying the originating order manager (105) of order confirmation.

NotifyOrderRejected (NotifyOrderRejectedData *) (reference 244 on Figure 2)

This method is called by ifMarketEventSink.NotifyOrderRejected() (276) when a market rejects an order. IfMarket.ReceiveMarketMessage() (233) calls ifMarketEventSink.NotifyOrderRejected() (276), which in turn calls ifExecutorEventSink.NotifyOrderRejected() (244), which operates by sending through a data communications network a message notifying the originating order manager (105) of the rejection.

NotifyOrderExecuted (NotifyOrderExecutedData *) (reference 246 on Figure 2)

In the illustrated embodiment, this method is called by ifMarketEventSink.NotifyOrderExecuted() (278) when a market executes an order. IfMarket.ReceiveMarketMessage() (233) calls ifMarketEventSink.NotifyOrderExecuted() (278), which in turn calls ifExecutorEventSink.NotifyOrderExecuted() (246), which operates by sending through a data communications network a message notifying the originating order manager (105) of the execution.

NotifyOrderCancelled (NotifyOrderCancelledData *) (reference 250 on Figure 2)

In the illustrated embodiment, this method is called by ifMarketEventSink.NotifyOrderCancelled() (282) when a market cancels an order. IfMarket.ReceiveMarketMessage() (233) calls ifMarketEventSink.NotifyOrderCancelled() (282), which in turn calls ifExecutorEventSink.NotifyOrderCancelled() (250), which operates by sending through a data communications network a message notifying the originating order manager (105) of the cancellation.

NotifyOrderKilled (NotifyOrderKilledData *) (reference 252 on Figure 2)

In the illustrated embodiment, this method is called by ifMarketEventSink.NotifyOrderKilled() (284) when a market kills an order. IfMarket.ReceiveMarketMessage() (233) calls ifMarketEventSink.NotifyOrderKilled() (284), which in turn calls ifExecutorEventSink.NotifyOrderKilled() (252), which operates by sending through a data communications network a message notifying the originating order manager (105) of the killing.

NotifyExecutionReport (NotifyExecutionReportData *) (reference 248 on Figure 2)

In the illustrated embodiment, this method NotifyExecutionReport () (248) is called as a supplement to NotifyOrderExecuted() (246) when a market executes an order. The purpose is to provide additional information to the order manager (105). The additional information is the execution price and the counterparty. This additional step is needed because the additional information is not provided by the market immediately upon execution, but is provided later. The execution notification is provided in two stages, however, so that users have a quick response. Markets can take many seconds, that is, a very long time, to provide price and counterparty. ifMarket.ReceiveMarketMessage() (233) calls ifMarketEventSink.NotifyExecutionReport() (280), which in turn calls ifExecutorEventSink.NotifyExecutionReport() (248), which operates by sending through a data communications network a message notifying the originating order manager (105) of the execution price, number of shares executed, and the counterparty.

NotifyShutdownBegins () (reference 268 on Figure 2)

Called by ifSmartExecutorRequest.Shutdown() (226) which was called by ifOrderEventSink.Shutdown() (293), in the illustrated embodiment, this method provides notification to order managers that the smart executor is shutting down and that no new messages will be processed. New order messages from order managers are to be rejected and new cancellation messages from order managers are to be ignored. Originating order managers (105) will be notified of orders executed before shutdown is completed.

In the illustrated embodiment, this method, ifExecutorEventSink.NotifyShutdownBegins(), operates by sending through a data communications network a message to notify connected order managers (105) to send no new order requests. The only substantive data elements in the message are the Action field (signifying beginning market shutdown) and the market service ID (comprising the range of initial letters served by the market service and smart executor being shut down).

NotifyShutdownComplete () (reference 270 on Figure 2)

In the illustrated embodiment, this method provides notification to order managers that the smart executor has completed shutdown and that no more messages will be generated by it. Called by ifSmartExecutorRequest.Shutdown() (226), this method NotifyShutdownComplete () (270) operates by sending through a data communications network to order managers (105) a message that the market service is now completely disconnected from the markets, i.e., there will be no further executions of pending orders. (NotifyShutdownBegins () (268) already stopped delivery of new orders.) The only substantive data elements in the message are the Action field (signifying market shutdown completion) and the market service ID (comprising the range of initial letters served by the market service being shut down).

Smart Executor Startup Procedure

At startup time, as shown on Figure 5, a market service (108), that is, the member method ifOrderEventsink.Startup() (290), first can call (512, 516, 520, 524, 528, 532) constructors (502, 504, 506, 508, 510) for ifMarketEventSink , ifSmartExecutorRequest, SmartExecutor, ifExecutorEventSink, and ifMarket.

Only one class object is instantiated for ifMarketEventSink, ifSmartExecutorRequest, SmartExecutor, and ifExecutorEventsink. If the class SmartExecutor is implemented by inheritance from ifMarketEventSink and ifSmartExecutorRequest, then the market service may start up by calling only the constructor for SmartExecutor instead of the constructors for ifMarketEventSink and ifSmartExecutorRequest. One class object for ifExecutorEventsink is created.

Multiple specializations or derived instances of ifMarket are constructed (510), one for each market that can receive orders. The specializations of ifMarket are derived class objects, one for each market, each containing private member methods designed to work with a particular market. More specifically, the specializations of ifMarket each implement the requirements of the well-known communications protocol of a particular market.

The constructors (502, 504, 506, 508, 510) all return pointers to the class objects constructed (514, 518, 522, 526, 530, 534) for storage in the market service (108). The constructor call (516) to ifSmartExecutorRequest() passes as a parameter for storage in ifSmartExecutorRequest the pointer to the ifMarketEventSink class object.

After constructing the class objects for the smart executor, the ifOrderEventSink.Startup()

(290) makes multiple calls (602) to ifSmartExecutorRequest->RegisterMarket() (222) as shown on Figure 6, one such call for each ifMarket class object constructed. That is, ifOrderEventSink.Startup() (290) makes one RegisterMarket() (222) call for each market that can receive orders. Each RegisterMarket() (222) call passes as a parameter a pointer (610) to one of the constructed ifMarket class objects.

Each time it is called, the method RegisterMarket() calls (606) ifMarket->MarketName() (234) receiving as a return the name of the market stored in the subject ifMarket class object, thus allowing ifSmartExecutorRequest to retain a list of pointers and names for all ifMarket class objects in existence. In addition to calling MarketName() (234), each time RegisterMarket() (222) is called, it also calls (608) ifMarket->RegisterMarketEventSink() (236) passing as a parameter the pointer (612) to the ifMarketEventSink class object, the pointer being retained in storage in each ifMarket class object so that the ifMarket class objects can call member methods in ifMarketEventSink to report results of operations within each ifMarket class object.

At this point in startup processing, the market service has created the class objects for the smart executor, connected them with pointers, and named the ifMarket objects. Next the market service will officially begin smart executor operation by use of a call to ifSmartExecutorRequest->Startup(), as shown on Figure 7.

As shown in Figure 7, IfSmartExecutorRequest->Startup() (224) is called (902) from a market service (108), ifOrderEventSink.Startup() (293), to initialize smart executor operation. ifSmartExecutorRequest->Startup() is called after RegisterMarket() (222 on Figures 2 and 6) so that the smart executor object ifSmartExecutorRequest now possesses pointers to all ifMarket class objects, one for each available market.

ifSmartExecutorRequest.Startup() functions by first issuing multiple calls (904) to ifMarket->Startup() (238), one such call for each existing ifMarket class object. Each

instance of ifMarket->Startup() establishes and maintains a data communications link (910) with the order router (112), whose operations and structure are described below. In the process of establishing the data communications link with the router, ifMarket->Startup() determines and stores in a member data element the data communications parameters needed by all the member functions of ifMarket for them to communicate with the router. If, for example, the data communications technology in use is tcp/ip, then ifMarket->Startup() will retain in storage the destination socket parameters for the router.

In certain embodiments, ifMarket->Startup() in calls (912) ifMarket.ReceiveMarketMessage() (233). ReceiveMarketMessage() (233) then, in blocking embodiments, can run generally continuously, in foreground or background, using the previously stored data communications parameters, by repeatedly requesting to receive and subsequently receiving data communications packets (914) from the order router (112). In embodiments which use, as an alternative to blocking waits for messages, non-blocking data communications calls, ifMarket.ReceiveMarketMessage() is identified by ifMarket.Startup() as a callback function for receipt of messages from a data communications API.

Packets returning from the router to be received by ReceiveMarketMessage() represent responses to orders and cancellations. Upon startup, therefore, there are no packets to be received from the router because no orders or cancellations have yet been sent to the router. ReceiveMarketMessage(), however, in this kind of embodiment, is now prepared to receive responses as soon as orders begin to flow.

After calling each existing specialization of ifMarket->Startup(), ifSmartExecutorRequest.Startup() calls ifExecutorEventSink.NotifyStartup() to notify (908) the order managers (105) that the smart executor is fully operational, ready to receive orders.

Smart Executor Example Operations

Referring generally to Figure 9: In a typical cycle of order operation for a smart order, processing begins when ifOrderEventSink.ReceiveOrderMessage() receives a message from an order manager (105), of the form shown in Figure 3, in which the Action field contains a type code indicating that the message comprises a new order. ifOrderEventSink.ReceiveOrderMessage() calls ifSmartExecutorRequest.RequestCreateOrder() passing as a parameter a pointer to the order.

RequestCreateOrder() stores the order in the order database and calls smartNewOrder(). smartNewOrder() initializes the Order->TryMarket index field to indicate the first market in the OrderedMarket array. The OrderedMarket array stores the eligible market identifiers in sequence, sorted according to a chosen sorting principle. smartNewOrder() then sets the number of scanned markets to zero, Order->ScannedMarkets = 0. Many implementations of the invention will keep track of the number of markets to which an order has been submitted. That number is typically stored in Order.ScannedMarkets. smartNewOrder() sets the order price to the current inside price and calls smartContinueOrder().

smartContinueOrder() contains the algorithms for deciding which market is to receive the order. smartContinueOrder() implements in this aspect of the invention an acyclic opportunity for the order to be submitted repeatedly to markets until the order times out, is cancelled, is submitted to all IOC markets at the inside price, or becomes unmarketable. If the order becomes unmarketable, smartContinueOrder() sends the order to a default market with the order's remaining time in force and original price. If the order is eventually submitted to all IOC markets at the inside price without being filled, the order is then submitted to a non-IOC market, such as SOES or SelectNet, again with

the order's remaining time in force and original price.

Smart Executor Shutdown Procedure

IfSmartExecutorRequest->Shutdown() (226) is called by the market service (108), as shown on Figure 8. Shutdown() will cleanly shut down the smart executor by canceling all pending orders and rejecting all new orders. When there are no outstanding orders in a market, Shutdown() will disconnect from each market. If a market fails to respond to cancellations after 30 seconds, Shutdown() will disconnect regardless. After shutdown has been performed, Startup() must be called before any additional orders may be processed. Shutdown() functions as described below.

First, Shutdown() calls (1002) ifExecutorEventSink.NotifyShutdownBegins() (268) to notify (1004) the order managers (105) to send no new order requests. Order managers (105) and the market services may continue to receive executions of pending/floating orders until shutdown is completed.

Second, Shutdown() loops (1006) through the order database (118), and issues cancellations by calling (1008) ifMarket->CancelOrder() (232) for each order with Order.isFloating = true.

Third, Shutdown() waits for a set period of time to allow markets time to confirm cancellations. Confirmations of cancellations will be effected by calls from the router (112) to ifMarket->ReceiveMarketMessage() (233), which will in turn call ifMarketEventSink.NotifyOrderCancelled() (282), one such call for each order cancelled. It is still possible to execute an order during this period if the order is executed in a market before the market receives the cancellation. Notifications from markets of orders executed is through calls (1016) to ifMarketEventSink.NotifyOrderExecuted() (278). Either result is acceptable so long as the orders are no longer floating.

Fourth, Shutdown() calls (1018) ifMarket->Shutdown() (240) once for each existing ifMarket class object to disconnect from all of markets.

Fifth, Shutdown() calls (1020) ifExecutorEventSink.NotifyShutdownComplete() (270) to tell the order managers (105) that the market service is now completely disconnected from the markets, i.e., there will be no further executions of pending orders.

At this point the market service (108) or the order managers (105) can be shut down, but whether they are shut down or not, the smart executor is now inoperative, no new orders coming in or going out, no pending orders executed or cancelled.

Order Router Detailed Structure

The order router functions primarily by receiving and administering data structures referred to as "router packets." We refer to the order router as "the order router" or "the router." The router includes two principal software processes and two principal data structures. The two principal processes include one process called the "router message processor" or "RMP" for receiving and sending router packets. The second principal process, called the "router SQL processor" or "RSP", is for storing data representations of orders in a SQL database, a method of creating an audit trail or a transactions log.

The two principal data structures include one data structure for order packets and another data structure, called a "router connection" or "RCT," to represent and store data describing data communications connections from the router to the market services and order ports. Although data communications connections are established in certain embodiments of the inventions by use of any generally available technology, the embodiments described in detail here generally implement communications connections implemented with tcp/ip and the Windows-Intel API known as "winsock." User skilled

in the art will recognize that other embodiments utilize other APIs including for example Berkeley sockets and the System V API known as the "Transport Layer Interface" or "TLI." Users skilled in the art will recognize that other embodiments of the invention utilize data communications protocols other than tcp/ip. User skilled in the art will recognize that other embodiments of the invention utilize data communications middleware.

The router sends and receives router packets to and from market services and to and from order ports. In all uses of router packets, the router packets utilize the same overall data structure. Order ports are additional software modules used to convert conventional data structures representing orders into the exact structure required for each market. The invention implements one order port for each market utilized by the smart executor.

There are two principal types of router packets. One type of router packet is used to establish a connection between the order router and market services or order ports and is referred to as a "hello packet." The second order packet type represents orders and is referred to as an "order packet." The same router packet data structure is used for both sending and receiving orders and responses to and from market services and to and from order ports.

An example of data structure for router packets is shown in Figure 13. The packet header (5002) stores indications of source or destination, type, and action. These three kinds of information in some embodiments of the invention are structured as separate variables if the header were implemented as a structure. Alternatively, the three kinds of information in some embodiments of the invention are bit-mapped into the single variable shown as the header (5002) in Figure 13 by performing a logical 'or' on three sets of numbers arranged as shown in Figure 17. In Figure 17, the C-style defines indicated by reference numbers 5400-5408 are used in some embodiments to denote the action represented by the packet, such as, for example, hellos, errors, rejections, confirmations, and so on. The

defines indicated by reference numbers 5409-5416 are used in some embodiments to denote the packet type, such as, for example, orders or cancellations. The defines indicated by reference numbers 5417-5432 are used in some embodiments to denote the source or destination of the packet, such as, for example, market services or order ports associated with markets such as SOES, SelectNet, Island, and so on.

The router packet data structure (5000), shown in Figure 13, also contains a C-type union (5008) used to store an additional structure representing data for hellos or for orders. The hello structure (5004) is further illustrated in Figure 14. Hellos contain a connection identification code called a Registration (5102) indicating whether the new connection is for a market service or an order port, Service codes (5104) which indicate for order ports the identity of the market served by the order port, and a ClientID field (5106) containing a short nickname for order ports and an indication of the range of symbols administered for market services.

Additional detail for the order union in the router packet structure (5006 on Figure 13) is shown in Figure 15. Figure 62 shows an example structure that can comprise the union referenced as 5006 on Figure 13. The order structure (5006) as illustrated in Figure 15 includes the following fields used as described below:

Date (5202) – the entry date of the order;

Time (5204) – the entry time for the order;

MarketID (5206) – the identification code for the market to which the order is directed; example values for the MarketID field are shown in Figure 18;

Action (5208) – an indication of the action represented by the packet; example values for the Action field are shown in Figure 19;

BrSeq (5210) – representing "Branch Sequence," a unique identifier for the order; this identifier will be used in all packets representing changes or responses related to the same order;

Side (5212) – representing Sell, Buy, or Sell Short;

Symbol(5214) – the market symbol for the security or stock to be bought or sold through the order;

Quantity (5216) – the quantity of shares affected by the action of the packet; that is, the original quantity ordered if the packet action is an original order, the quantity bought or sold if the packet action is an execution, the quantity of shares rejected if the packet action is a rejection, and so on;

Remaining (5218) – the quantity of shares not affected by the action of the packet; that is, if the original quantity was 100 shares on the sell side of an order packet, and the current packet action shows an execution with quantity set to 60, Remaining will be set to 40;

Price (5220) – contains the order price for limit orders, "MKT" or "0" for market orders;

Union (5228) – a union reserving memory storage space for order detail specific to particular markets, the invention including separate union structures defined for each market connected to the router, example names for typedefs and structure names for actual markets being illustrated at reference numbers 5230-5250 and 5254-5274 on Figure 15 with a general representative example (5254, 5276) illustrated in further detail on Figure 16.

Figure 16 shows a general example called example_market_order (5302), directed to an example market rather than any particular actual market, of the detail structure of the packet union used to store the order detail needed for particular markets. Included in Figure 16 is an example C-style typedef (5314) illustrating how the example data structure (5252) is defined for use in a union. The example (5302) includes the following fields used as described below:

Minimum (5304) – the minimum number of share to be executed, zero indicating

that any number is allowed, an amount equal to the order quantity indicating that the execution is to be all or nothing;

TIF (5306) – time in force, the period stated in seconds for which the order remains valid;

Display (5308) – instructions to the destination market for public display of the order, typical values being "HIDDEN," "SUBSCRIBER," and "NORMAL;"

Reference (5310) – unique order identification code assigned by the market in which the order is pending;

Contra (5312) – identification code for the party on the other side of an order.

An example of the router connection data structure (5702) is shown in Figure 20. The router connection data structure in the example (5702) includes the following fields used as described below:

Socket (5704) – the socket identification returned by the API or operating system when the related data communications socket is created.

Registration (5708) – an indication whether the connection is for a market service or an order port.

Service (5710) – for connections to order ports, this field contains an identification code for the market associated with the order port; the codes in certain embodiments, for example, are those shown in Figure 17, reference numbers 5417-5426.

ClientID[5] (5712) – for connections to market services, an array of characters indicating the range of securities symbols administered by the market service.

Packets (5712) – a pointer to the packet queue associated with the RMP associated with a router connection data structure (5702).

In one embodiment, the router is implemented as shown on Figure 21 with multiple instances of router message processors (5814, 5838) running in multiple threads, each of the router message processors (5814, 5838) having associated with it one router connection data structure or RCT (5844, 5846), each router message processor (5814, 5838) being instantiated to serve and being associated with one particular market service (5802) or order port (5830). In addition, each router message processor (5814, 5838) has associated with it a queue (5820, 5822) implemented as a linked list containing packets waiting to be sent to the particular market service (5802) or order port (5830) associated with the router message processor (5814, 5838).

Order Router Operations

Bearing in mind the overall structure just described, the general operation of an RMP is described for one embodiment of the invention, shown on Figure 21, as looping repeatedly through the following steps:

receiving (5812, 5834) across a data communication network (5804, 5832) a packet from the market service (5802) or order port (5830) with which the RMP (5814, 5838) is associated;

attaching (5818, 5842) that packet to the end of the packet queue (5820, 5822) associated with the RMP (5814, 5838) for the destination market service (5802) or order port (5830) identified in the packet, if the associated packet queue (5820, 5822) exists;

attaching (5854) that packet to the end of the queue associated with the source market service (5856) for return to the market service (5856) and setting the

Action field in the packet data structure to an error code, if the packet is to be routed to a destination order port (5830) and the associated packet queue (5822) for the order port does not exist;

attaching (5852) that packet to the end of a default queue (5856), if the packet is to be routed to a destination market service (5802) and the associated packet queue (5820) does not exist;

attaching (5816, 5840) a copy of the packet to the end of the SQL database queue (5824);

retrieving (5848, 5850) a packet from the packet queue (5820, 5822) associated with the RMP (5814, 5838);

sending (5836, 5812) across a data communication network (5804, 5832) the retrieved packet to the market service (5802) or order port (5830) with which the RMP (5814, 5838) is associated;

checking (5848, 5850) the packet queue (5820, 5822) associated with the RMP (5814, 5838) to determine whether it contains a packet to be sent to the associated market service (5802) or order port (5830);

sending (5836, 5812) the first waiting packet, if there is one, from the associated packet queue (5820, 5822) to the market service (5802) or order port (5830) associated with the RMP (5838, 5814); and

removing the sent packet from the packet queue (5820, 5822).

An associated packet queue (5820, 5822) might not exist if, for example, its associated RMP (5814, 5838) were not running, had not been started, or had failed for some reason,

any reason. Attaching market service packets to the default queue (5852) is robust because each RMP for a market service upon startup, in addition to the processing described above, also scans the default queue (5860) to determine whether the default queue contains packets waiting for the startup RMP (5814), and, if there are any such packets, transfers them to the RMP's associated queue (5820).

Moreover, to increase robustness, as illustrated in Figure 22, the default queue (5856), in some embodiments, as shown on Figure 22, is implemented as a persistent queue mirrored in background so that even computer crashes cannot lose the default queue. That is, a parallel mirror process or thread (5902) can periodically read the entire default queue (5906) and write (5910) it to disk (5904). Upon restart following a failure, the mirror process (5902) can read from the disk (5904) the packets comprising the default queue (5912) and write them back to the default queue in volatile memory (5908). The same mirror process or thread (5902) is used in this embodiment also, as shown on Figure 22, to backup (5916, 5910) and restore (5912, 5914) the SQL database queue (5824).

In this embodiment of the invention, as shown in Figure 21 and Figure 22, in summary, it can be seen that no packet is ever lost for any reason. If a packet's destination queue does not presently exist for any reason, the packet is attached to the default queue (5856) or returned rejected with an error message(5854). While attached to the default queue (5856), packets are robustly protected from loss (5902, 5904). When a packet's destination queue comes again into existence, the packet is transferred to the destination queue and continues on its way (5860). All packets are copied to a SQL database (5862, 5826, 5828). The packet queue for the SQL database is robustly mirrored to persistent storage (5902, Figure 22), assuring that all packets eventually find their way to the SQL database. The SQL database forms a complete transactions log from which all transactions can be recovered in the event of a failure. The particular transactions recovery procedure to be employed can be any of a number of known methods. The embodiment illustrated on Figure 21 and 22 uses a SQL database to implement persistant

storage of a log of transactions. Other embodiments use other means of implementing persistant storage of a log of transactions.

Referring again to Figure 21: The steps of receiving (5812, 5834) and sending (5836, 5812) packets across data communications networks (5804, 5832) are accomplished through the use of known API calls. For increased efficiency, for embodiments of the invention using the winsock API, the step of receiving a packet (5812) is preceded by a call to the winsock select() function which will return an indication whether there are any packets waiting to be received. The advantage of including this step is that select() typically returns control to the calling process much faster than the winsock functions associated with actual retrieval of tcp/ip messages. Similar efficiencies may be available from analogous calls supported by other APIs.

Referring again to Figure 21, the operation of RSP (5826), the router SQL processor, the second principal process in the router, is described as follows. The RSP is a single-instance thread or process having a persistent connection to a SQL server database (5828). The only purpose of the RSP (5862) is to retrieve packets from its queue (5824) and write them to a SQL database (5828). The RSP operates in a continuous loop in which it first checks for the presence of a packet in its queue (5824) and, if there is a packet present in the queue, retrieves the packet and writes it to the SQL database (5828). The RSP then loops back to check the queue again, and so on. In this embodiment of the invention, the RSP is implemented as a separate thread or process because writing orders to a SQL database is relatively slow compared to the speeds needed for order routing. Implementing the RSP in parallel with the other router functions minimizes the impact of SQL database storage on the quickness of the order routing operations.

Figure 21 illustrates the overall flow of normal order routing operations. An order packet is received (5810) from a market service (5802) by an RMP (5814). The order packet field MarketID (5206 on Figure 15) identifies the order port (5838) to which the order

packet is to be routed. (Figure 21 illustrates for purposes of explanation only two sets of queues and RMPs, one each for a market service and an order port. In actual implementations of the invention, however, there will be multiple queues and RMPs for order ports and for market services.) Assuming for illustration of normal operation that the queue for the order port exists, the packet is attached (5818) to the end of the queue (5822). When the packet works its way to the top of the order port queue (5822), the RMP for the order port (5838) removes the packet from the queue (5850) and sends it 5836) to the order port (5830). The order port formulates an order according to its associated market protocol and sends the order to the market, eventually in normal course of processing receiving a response. The response is fashioned by the order port into a responsive order packet. In this case the order packet action field will indicate that the responsive packet represents an order execution. The order port (5830) sends (5834) the order packet back to the RMP (5838) for the order port. The order port's RMP reads the destination market service from the packet header and attaches (5842) the packet to the end of the queue (5820) for the destination market service (5802). When the packet works its way to the top of the queue, the RMP (5814) for the market service removes the packet from the queue (5820) and sends (5812) it to the market service (5802), thus completing one typical operational cycle of routing for a packet representing an order and a market response to the order.

Order Router Startup

The router in the embodiment under discussion is started by use of a continuously looping process called a "connection processor." The connection processor (6004) functions, as shown on Figure 23, by requesting (6010) by known means a tcp/ip connection from an API (6006). When a connection is available, the API notifies (6012) the connection processor of the connection, and at the same time passes the connection processor the socket identification for the connection. The connection processor then creates (6014) an RCT (6008), a router connection data structure, at the same time storing

the socket identification in the RCT (reference number 5704 on Figure 20) and retaining (6016) a pointer to the RCT. The connection processor then starts (6018) an RMP (6002), a router message processor, passing the RMP the pointer to the RCT, thus enabling the RMP to make API calls using the socket identification stored in the RCT. The connection processor then returns to the first step, requesting (6010) a connection from the API (6006).

At startup, therefore, the connection processor will continue to request connections and establish RCTs and RMPs until one RCT and one RMP has been established for each market service and for each order port seeking to connect to the router. Using this procedure at startup will promptly connect all of the market services and order ports intended to function with the router. By continuing throughout the trading day to seek connections, the connection processor also will promptly reconnect any market service or order port that is temporarily disconnected for any reason. The connection processor also will promptly connect any new market service or order port, not present at startup, that comes on-line during the trading day.

In this aspect of the invention, the market service and order port are programmed to transmit hello packets as their first transmissions after startup. As shown in Figure 14, a hello packets identifies for the RMP receiving it the registration (5102), the service (5104), and the clientID (5106) for the market service or order port to which the RMP is connected. The RMP reads the registration (5102), service (5104), and clientID (5106) from the hello packet and writes the contents of those fields to the parallel fields in the router connection data structure (5702), references 5708, 5710, and 5712 respectively on Figure 20. In summary, the startup process as described establishes a connection to a market service or order port, creates a data structure to represent the connection, starts an RMP thread or process to administer the connection, advises the RMP whether the RMP is connected to a market service or an order port, and further advises the RMP how to identify that market service or order port for purposes of data communications. The latter

step of identification for purposes of data communications, in the example tcp/ip protocol, means recording a socket identification (5704) in the field reserved for it.

Order Port Operations

In one aspect of the invention, as shown on Figure 1, the order ports are connected through data communications networks (132, 134) to an order router (112) and a market (114). Each order port (128) is responsible for communications of orders, cancellations, and responses between one order router (112) and one market (114), as shown on Figure 12, although, an order router (112) can administer orders, cancellations, and responses to and from multiple order ports (128).

Upon startup, the order port registers with its associated order router by sending the router a hello message in the form shown in Figure 13, which by union includes the structure shown in Figure 14. The order port also establishes through known means a data communications connection to the market with which the order port will be associated. Recall that each order port connects to only one market.

During normal operations, the order port (128) functions by receiving order and cancellation messages from the order router (112) in the form shown in Figure 13, which by union includes the structure shown in Figure 15. The order port (128) will also receive response messages from the market in the form dictated by the particular protocol in use by the market with which the order port is associated. Market communications protocols are well-known public information. Upon receiving an order or cancellation message, the order port (128) converts the structure of the message into the form of order or cancellation required by the particular market protocol of the market to which the order port is connected and sends the order or cancellation to the market (114). Upon receipt of a response from the market (114), the order port converts the response into the

form shown in Figure 13, which by union includes the form shown in Figure 15, and sends the response to the order router (112).

Data Communications Middleware and Data Communications Frameworks

The example embodiments described in detail above, generally implement data communications by use of direct calls to APIs such as winsock or Berkeley sockets. Other embodiments of the inventions replace direct calls to APIs with calls to installations of data communications middleware. Data communications middleware is a software layer forming a programming interface designed to operate the applications programming interface ("API") layer of data communications networks by providing a generic interface to them. In an internet network, for example, the middleware operates just above the layer comprising the API, which in turn is layered just above the transport layer known as the transmission control protocol, "tcp" or "tcp/ip."

One advantage of middleware is that middleware provides a common programming interface for use software applications for data communications, regardless of the underlying communications API. That is, in embodiments of the invention utilizing middleware installed in the Windows Intel environment, middleware provides an interface to Microsoft's "winsock." When embodiments of the invention originally implemented for Windows Intel are subsequently implemented on Unix systems, the middleware is reconfigured to interface with any Unix-oriented API, including for example Berkeley sockets or the System V API known as the Transport Layer Interface or TLI. Persons skilled in the art will realize that the reconfiguration from Windows Intel to Unix, using middleware, require no changes in the applications layer. The only changes required are in the underside of the middleware, changing winsock calls to, for example, Berkeley sockets calls. Embodiments using third-party middleware products already ported from Windows Intel to Unix, require no additional programming to port such embodiments from the Windows Intel environment to Unix or to other environments

or platforms.

Some embodiments of the invention are configured for message-oriented middleware. Although message-oriented middleware is generally expected to be faster than remote procedure calls, some embodiments of the invention are implemented by use of remote procedure calls. Similarly, the data communications aspects of the invention in some embodiments are implemented using communications middleware object request brokers developed under the Common Object Request Broker Architecture or "CORBA," the standard for interoperability developed by the nonprofit organization known as the Object Management Group of Framingham, Massachusetts. In addition to a programming interface, middleware, in some embodiments of the invention, also provides transactions logging in support of functional fault tolerance and robustness.

In addition to data communications by means of networks as described above, data communications among elements of the invention in some embodiments is implemented by use of electronic or optical transmissions of data, local area networks, wide area networks, intranets, internets, dedicated lines, satellite links, optical links, and other types of communications. Moreover, although data communications among the customer workstations, order managers, market services, order routers, and order ports in some embodiments is implemented through data communications links, an approach having certain advantages with respect to extensibility, robustness, and scalability, other embodiments of the invention achieve data communications among elements by other means including, for example, implementation of, for example, market services and order routers on the same computer utilizing compilation against shared libraries or dynamically linked libraries supporting shares data structures and direct calls among functions, subroutines, and member methods, with no need for intervening tcp/ip data communications links or other kinds of intervening data communications links.

APPENDIX II

Detailed Description And Figures to: U.S. Patent Application Serial No. 09/578,947 entitled "Solutions Server" filed May 25, 2000, and *incorporated herein by reference.*

Turning now to Figure 1, a first aspect of the invention is seen, a method of providing solutions to problems (12). One illustrated embodiment provides for generating problem definitions (18) for problems (12), receiving environmental information (16) required for generating solutions for the problems, generating solutions (19) for the problems dependent upon the environmental information and the problem definitions, and communicating solutions (24) to clients (26) before the solutions are needed. Although the invention can be applied to a wide range of problems, in certain embodiments, the generated problem definitions are for problems related to securities trading.

An example embodiment, directed to generating solutions for securities trading, is shown in Figure 1A, in which generating solutions (19) is accomplished in dependence upon environmental information (10) comprising quotes (2002) from securities markets, hidden quantity ratios (2016), and latencies (2014). In many such embodiments, solutions include solution quantities calculated dependent upon quote quantities and hidden quantity ratios. In many such embodiments, solutions comprise solution records sorted or indexed according to latencies for markets identified by MPIDs in the solution records. In the embodiment shown in Figure 1A, generated solutions are stored (30) in solutions records (1906) in a database (22).

A further embodiment shown in Figure 1 includes storing problem definitions (29) in problem definition records in a database (22). The problem definition records (314) in many embodiments are disposed in the database (22) as shown on Figure 2. An example structure for problem definition records is shown as reference (314) in Figure 2G.

A further embodiment illustrated in Figure 1 includes storing the solutions (30) in solutions records in the database (22). The solution records (316) in many embodiments are disposed in the database (22) as shown on Figure 2. An example structure for

solutions records is shown as reference (316) in Figure 2F. In typical embodiments, the solutions records (316) have at least one relation (315) to the problem definition records (314), one example of which is shown on Figure 2.

A still further embodiment shown on Figure 1 includes retrieving (32) at least one solution from the solutions records in the database (22). In some embodiments illustrated by Figure 1, at least one client (26) is a broker-dealer computer system programmed and operated to effect securities trading.

Some embodiments illustrated by Figure 1, when received environmental information (10) changes, include generating additional solutions (19) dependent upon the changed environmental information and the problem definitions and communicating (24) additional solutions to clients before the additional solutions are needed. In many embodiments, problems are categorized according to type, an example of which is the use of the probdeftype field (202) in the problem definition structure (314) shown in Figure 2G.

In a still further embodiment, as shown on Figure 1, the invention includes generating subscriptions (20) for solutions, the subscriptions comprising relations between clients and types of problems. An example of an embodiment implementing a relation between clients and types of problems is the subscription structure (310) in Figure 2D, where the subscription comprises data elements identifying a client (216) and a problem type (202).

Referring to Figure 3, a still further embodiment of the invention is seen. One embodiment shown in Figure 3 includes submitting (302) a request (304) for a subscription. A still further embodiment illustrated at Figure 3 provides for creating (308) a subscription record (310) in response to the submission (302) of the request (304) for a subscription. A typical example of a subscription record (310), shown at Figure 2D,

comprises data elements identifying a client (216) and a problem type (202).

A still further embodiment illustrated at Figure 3 provides for creating (312) at least one problem definition (314) record dependent upon problem definition rules (306). An example problem definition data structure used in many embodiments directed to securities trading is illustrated in Figure 2G as comprising problem definition type (202), side (208), symbol (210), and quantity (226). An example problem definition rule structure used in many embodiments directed to securities trading is illustrated in Figure 2B as comprising problem definition type (202) and quantity (204).

Turning now to Figure 4, a further embodiment of the invention is seen. One embodiment illustrated at Figure 4 includes receiving environmental information which in the illustrated embodiment comprises market information (402) in the form of quotes (28), the quotes (28) comprising data elements representing side (208), symbol (210), quantity (212), market (206), and a tag (214) as shown in Figure 2C. The tag (214) is an indication of the status of the quote, including, for example, whether the quote is an open quote or a closed quote.

In a further embodiment illustrated at Figure 4, the invention provides, when a quote is received and the quote tag does not indicate (413) that the quote is closed, finding a problem definition record (404) having the same side and symbol as the quote. In a still further embodiment shown in Figure 4, the invention provides for searching for a solution record (408) having the same problem type, side, and symbol as the problem definition record having the same side and symbol as the quote and the same market as the quote.

When the solution record is found (409), the illustrated embodiment includes updating the solution record (410) with the price from the quote. When the solution record is not found (411), the illustrated embodiment includes creating a new solution record (412) having the same problem type, side, and symbol as the problem definition record having

the same side and symbol as the quote, the same market as the quote, and the same price as the quote. When a quote is received and the quote tag indicates that the quote is closed (415), the illustrated embodiment includes deleting (414) solution records having the same side, symbol, and market as the quote.

An additional example embodiment, directed to solutions for securities trading, is shown in Figure 4A, in which creating solutions (412) is accomplished in dependence upon quotes (2002) from securities markets, hidden quantity ratios (2016), and latencies (2014). In many such embodiments, solutions include solution quantities calculated dependent upon quote quantities and hidden quantity ratios. In many such embodiments, solutions comprise solution records sorted or indexed according to latencies for markets identified by MPIDs in the solution records. In the embodiment shown in Figure 4A, generated solutions are stored (30) in solutions records (1906) in a database (22).

Turning to Figure 5, a still further embodiment of the invention is seen. One embodiment shown in Figure 5 includes repeatedly finding (502) a subscription record so that each existing subscription record (310) is found in turn. When a subscription record (504) is found, the illustrated embodiment includes finding (508), for each found subscription record (510), a related record of data communications parameters (516) for the client identified in the found subscription record (504).

For each found subscription record (504), the illustrated embodiment includes finding (506) at least one related problem definition record (510). When at least one problem definition record (510) is found, the embodiment of Figure 5 includes finding (512), for each found problem definition record (510), at least one related solution record (514). A still further embodiment, as shown on Figure 5, provides for communicating (518, 520), dependent upon data communication parameters identified in the found record of data communications parameters (516), to the client (26) identified in the found subscription

record (504) at least one data element of the found solution record (514).

Turning to Figure 6, still further embodiments of the invention are seen. One embodiment illustrated at Figure 6 includes communicating solutions (24) to clients as communicating solutions to an order processing system (26) on a broker-dealer computer. A further embodiment shown on Figure 6 includes receiving (604) at least one customer order (606). The customer order, illustrated in detail at reference 606 on Figure 2H, comprises data elements identifying symbol (210), quantity (228), and optionally, price (230), market (224), and order type (226). In many embodiments, the customer order type has a relation to a problem type. Problem types as used in typical embodiments are illustrated at reference (202) in solution structure (316) on Figure 2F. A still further embodiment illustrated in Figure 6 includes finding (608) at least one solution record (610) having the same symbol (210) as the received customer order (606) and also having a problem type related to the customer order type (226) in the received customer order (606).

A still further embodiment illustrated in Figure 6 includes sending (612) to at least one market (614) at least one solution order (616), the solution order being dependent upon the customer order and the data in the found solution record (610). A still further embodiment illustrated in Figure 6 includes sending (612) to at least one market (614) at least one solution order (616), the solution order comprising the side, symbol, quantity, price, and market data elements from the found solution record (610).

Turning to Figure 7, a further aspect of the invention is seen, that is, a system (702) for providing solutions to problems. One embodiment shown in Figure 7 includes means for generating problem definitions (704) for problems, means for receiving environmental information (706) required for generating solutions for the problems, means for generating solutions (708) for the problems dependent upon the environmental information and the problem definitions, and means for communicating solutions (710) to

clients before the solutions are needed.

The means for generating problem definitions (704) in typical embodiments is a computer programmed to store problem definitions in a data structure in computer memory. An example of such a data structure is provided at reference (314) in Figure 2G. The means for receiving environmental information (706) in typical embodiments is a computer programmed to receive a data stream through a data communications port, typically, not always, connected to a network. An example of such a data stream is a Nasdaq feed, a stream of ticker information or quotes provided to subscribers by Nasdaq.

The means for generating solutions (708) for the problems dependent upon the environmental information and the problem definitions in typical embodiments is a computer programmed to generate and store solutions in a data structure in computer memory. An example of such a data structure is provided at reference (316) in Figure 2F.

The means for communicating solutions (710) to clients before the solutions are needed, in typical embodiments, is a computer programmed to retrieve solutions from computer memory and transmit through a communications port, often over networks, to clients. In other embodiments, at least one of the clients is closely coupled to the system for providing solutions, with communicating solutions structured through shared memory, software subroutine calls, or calls to member functions in class objects.

In some embodiments clients (712) are implemented in the same overall computer system (702) as the system (702) for providing solutions to problems. Other embodiments have clients (714) as separate entities. Still other embodiments have other relations between the system for providing solutions and clients, all which relations are within the scope of the present invention.

In some embodiments implemented as shown in Figure 7, the problem definitions are for problems related to securities trading. A further embodiment provides means for storing the problem definitions (716) in problem definition records in a database (22).

A still further embodiment provides means for storing the solutions (718) in solutions records in the database (22). The means for storing the problem definitions and means for storing solutions, in the illustrated embodiment, is computer memory coupled to a processor. The computer memory has various forms in various embodiments, including random access memory, magnetic disk drives, read only memory, programmable read only memory, and erasable programmable read only memory. Means for storing takes many forms in computer memory, all within the scope of the invention.

In many embodiments, the solutions records have at least one relation to the problem records, for example, one-to-many or many-to-many. Other relations are used in other embodiments, all within the scope of the invention. A still further embodiment provides means for retrieving at least one solution (720) from the solutions records in the database. Means for retrieving in typical embodiments is a computer processor coupled to computer memory and programmed to search the memory. In some embodiments, at least one client (712, 714) is a broker-dealer computer system programmed and operated to effect securities trading.

A still further embodiment illustrated on Figure 7 provides for use when received environmental information changes means for generating additional solutions (722) dependent upon the changed environmental information and the problem definitions and means for communicating additional solutions (724) to clients before the additional solutions are needed. In many embodiments of the kind illustrated on Figure 7, problems are categorized according to type.

A still further embodiment shown in Figure 7 provides means for generating subscriptions (726) for solutions, the subscriptions comprising relations between clients and types of problems. Means for generating subscriptions in such embodiments include available data entry screens for entering data into a computer, the data being received by a computer processor capable of storing the subscriptions in computer memory.

Turning now to Figure 8, a further aspect of the inventions is seen. Figure 8 shows an embodiment providing means for submitting a request (804) for a subscription. Means for submitting a request for a subscription in such embodiments include available data entry screens for entering data into a computer, the data being received by a computer processor capable of storing the subscriptions in computer memory.

A further embodiment shown on Figure 8 provides means for creating (806) a subscription record in response to the submission of the request for a subscription. Means for creating a subscription record in such embodiments include a computer processor programmed to create and store the subscription record in computer memory. In many such embodiments, the subscription record comprises data elements identifying a client and a problem type.

A still further embodiment shown on Figure 8 provides means for creating (808) at least one problem definition record dependent upon problem definition rules. Means for creating a problem definition record in such embodiments include a computer processor programmed to read rules from a computer memory store of problem definition rules and create and store a problem definition record in computer memory. In many such embodiments, the problem record comprises data elements identifying problem type, side, symbol, and quantity.

Turning now to Figure 9, a further embodiment of the invention is seen. An embodiment shown in Figure 9 provides means for receiving environmental information comprising

means for receiving (904) market information in the form of quotes, the quotes typically comprising, as shown in figure 2C, data elements of side (208), symbol (210), quantity (212), market (206), and a tag (214), the tag being a status code for the quote. Means for receiving market information, in embodiments similar to the ones illustrated in Figure 9, include ticker feeds, quote feeds, and market data feeds from Nasdaq and from other exchanges, as well as similar feeds from ECNs, market makers, other markets, and other broker-dealers.

A further embodiment of the invention, shown on Figure 9, provides, for use when a quote is received and the quote tag does not indicate that the quote is closed, means for finding (906) a problem definition record having the same side and symbol as the quote. Means for finding a problem definition record in such embodiments include a computer processor programmed to search through problem definition records in a table or database using established search criteria.

A further embodiment of the invention, shown on Figure 9, provides means for searching (908) for a solution record having the same problem type, side, and symbol as the problem definition record having the same side and symbol as the quote and the same market as the quote. Means for searching for a solution record in such embodiments include a computer processor programmed to search through solution records in a table or database using established search criteria.

A further embodiment, shown on Figure 9, provides, for use when a solution record is found, means for updating (910) the solution record with the price from the quote. Means for updating the solution record, in many such embodiments, is a computer processor programmed to write the updated price into a price field or data element in the solution record, data structure, or class object in computer memory.

A further embodiment, shown on Figure 9, provides, for use when the solution record is not found, means for creating (912) a new solution record having the same problem type, side, and symbol as the problem definition record having the same side and symbol as the quote, the same market as the quote, and the same price as the quote. In such embodiments, means for creating a new solution record typically comprise a computer processor programmed to create a data structure having defined data elements and write into those data elements the information from the quote and the problem definition record. An example of such a data structure is shown at reference 316 on Figure 2F. The example of Figure 2F is directed to solutions for problems of securities trading and includes data elements of problem type (202), side (208), symbol (210), quantity (220), price (222), and market identification code (224).

A still further embodiment shown on Figure 9 provides, for use when a quote is received and the quote tag indicates that the quote is closed, means for deleting (914) solution records having the same side, symbol, and market as the quote. Means for deleting solution records in such embodiments include a computer processor programmed to search through solution records in a table or database using established search criteria, locate records meeting the criteria, and delete those records from the table or database, or alternatively, mark the records as not in use.

Turning now to Figure 10, a further embodiment is seen to provide means for repeatedly finding (1004) a subscription record so that each existing subscription record is found in turn. Means for repeatedly finding a subscription record in such embodiments include a computer processor programmed to search through subscription records in computer memory, said computer memory including, in various embodiments, arrays, linked lists, linked lists of pointers to other structures, tables, and databases.

A still further embodiment, shown on Figure 10, provides means for finding (1006), for each found subscription record, a related record of data communications parameters for

the client identified in the found subscription record. Means for finding a related record of data communications parameters in such embodiments include a computer processor programmed to search through records of data communications parameters in computer memory, said computer memory including, in various embodiments, arrays, linked lists, linked lists of pointers to other structures, tables, and databases.

A still further embodiment, shown on Figure 10, provides means for finding (1008), for each found subscription record, at least one related problem definition record. Means for finding a related problem definition record in such embodiments include a computer processor programmed to search through problem definition record in computer memory, said computer memory including, in various embodiments, arrays, linked lists, linked lists of pointers to other structures, tables, and databases.

A still further embodiment, shown on Figure 10, provides, for use when at least one problem definition record is found, means for finding (1010), for each found problem definition record, at least one related solution record. Means for finding a related solution record in such embodiments include a computer processor programmed to search through solutions records in computer memory, said computer memory including, in various embodiments, arrays, linked lists, linked lists of pointers to other structures, tables, and databases.

A still further embodiment, shown on Figure 10, includes means for communicating (1012), dependent upon data communication parameters identified in the found record of data communications parameters, to the client identified in the found subscription record at least one data element of the found solution record. Means for communicating data from a solution record in such embodiments include data communications ports, networks, satellite links, dedicated phone lines, intranets, internets, and extranets coupling the search processor to at least one client.

Turning now to Figure 11, further embodiments of the invention are seen. Figure 11 shows one embodiment providing means for communicating solutions to clients further comprising means for communicating (1104) solutions to an order processing system on a broker-dealer computer. Means for communicating data from a solution record in such embodiments include data communications ports, networks, satellite links, dedicated phone lines, intranets, internets, and extranets coupling the search processor to at least one broker-dealer computer.

A further embodiment shown on Figure 11 provides means for receiving (1106) at least one customer order. The means for receiving an order in such embodiments typically includes customer workstations coupled to an order processing system, the coupling effected typically through data communications ports, networks, satellite links, dedicated phone lines, intranets, internets, and extranets coupling the search processor to at least one broker-dealer computer.

In such embodiments, the customer order typically comprises data elements identifying symbol (210), quantity (228), and optionally, price (230), market (224), and order type (226), as shown on Figure 2H. In many embodiments of the invention as it relates to processing systems for securities, the customer order type has a relation to a problem type, including, for example, the order type having a one to one correspondence with a problem type. All relations between order type and problems type are within the invention.

A further embodiment, shown on Figure 11, provides means for finding (1108) at least one solution record having the same symbol as the received customer order and also having a problem type related to the customer order type in the received customer order. Means for finding at least one solution record in such embodiments includes a computer processor programmed to search through solutions records in computer memory, said computer memory including, in various embodiments, arrays, linked lists, linked lists of

pointers to other structures, tables, and databases.

A still further embodiment, shown on Figure 11, provides means for sending (1110) to at least one market at least one solution order, the solution order being dependent upon the customer order and the data in the solution record. Means for sending the solution order in such embodiments include data communications ports, networks, satellite links, dedicated phone lines, intranets, internets, and extranets coupling the search processor to at least one market.

A still further embodiment, shown on Figure 11, provides means for sending (1112) to at least one market at least one solution order, the solution order comprising the side, symbol, quantity, price, and market data elements from the found solution record. Means for sending the solution order in such embodiments include data communications ports, networks, satellite links, dedicated phone lines, intranets, internets, and extranets coupling the search processor to at least one market.

Turning to Figure 12, a further aspect of the invention is seen, that is, a solutions server. One embodiment of a solutions server, illustrated at Figure 12, includes a processor (36) coupled to at least one source of environmental information (10) and coupled also to at least one client (26). The processor (36) is programmed, in one embodiment illustrated in Figure 14, to generate problem definitions (1304) for problems, receive environmental information (1302) required for generating solutions for the problems, generate solutions (1306) for the problems dependent upon the environmental information and the problem definitions, and communicate solutions (1308) to clients before the solutions are needed. The solutions server of Figure 12 includes also a memory (32) coupled to the processor (36), the processor being programmed also to store (1310) in the memory problem definitions and solutions. Although the solutions server aspect of the invention is useful for many different kinds of problems, many embodiments of the illustrated solutions

server include problem definitions and solutions related to securities trading.

A further embodiment of the solutions server of Figure 14 includes computer memory (32) in which is stored (1310) problem definitions and solutions. Many embodiments of the solutions server implement computer memory storing problem definitions in the form of a database (22) with problem definitions records in a table (314) as shown in Figure 2. An example of problem definitions records structured to address problems of securities trading is shown in Figure 2G as including data elements of problem type (202), side (208), symbol (210), quantity (226). An example of solutions records structured to address problems of securities trading is shown in Figure 2F as including data elements of problem type (202), symbol (210), side (208), quantity (220), price (222), and market identification (224).

A further embodiment illustrated at Figure 14 provides a processor (36) further programmed to retrieve (1312) at least one solution record from the solutions records in the database (22). In many embodiments of the aspect of the invention illustrated in Figure 14, at least one client (26) is a broker-dealer computer system programmed and operated to effect securities trading.

In many embodiments of the solutions server illustrated in Figure 12 and 14, problems are categorized according to type. An example of a data structure (314) defining a problem categorized according to problem type (202) is shown at Figure 2G.

A further embodiment illustrated at Figure 14 provides a processor (36) further programmed to generate subscriptions (1314) for solutions. In many embodiments of the kind illustrated, subscriptions comprise relations between clients (216) and types of problems (202) as shown in data structure (310) in Figure 2D.

Turning to Figure 15, additional embodiments of a solution server are seen. One embodiment shown in Figure 15 provides for a client's (26) submitting to a processor (36) a request for a subscription (1402). Another embodiment shown on Figure 15 provides the processor (36) programmed to create a subscription record (1404) in response to the submission (1402) of the request for a subscription. In some embodiments, for example, the subscription record (310) comprises data elements identifying a client (216) and a problem type (202) as shown on Figure 2D.

Still another embodiment illustrated on Figure 15 provides the processor (36) further programmed to create (1406) at least one problem definition record dependent upon problem definition rules. In some embodiments, for example, the problem definition record (314) comprises data elements identifying problem type (202), side (208), symbol (210), and quantity (226), as shown in Figure 2G.

Turning to Figure 16, further embodiments of a solution server are seen. Figure 16 illustrates an embodiment directed to securities trading in which environmental information comprises market information in the form of quotes received (1502) by a processor (36). An example of a data structure useful for this class of quotes (28), the quotes further comprising data elements identifying side (208), symbol (210), quantity (212), market (206), and a tag (214), is shown at Figure 2C.

A further embodiment of the solutions server, shown in Figure 16, provides the processor being further programmed to find (1504), when a quote is received (1502), a problem definition record having the same side and symbol as the quote. A further embodiment of the solutions server, shown in Figure 16, provides the processor further programmed to search (1506) for a solution record having the same problem type, side, and symbol as the problem definition record having the same side and symbol as the quote and the same market as the quote. A still further embodiment of the solutions server, shown in Figure 16, provides the processor being further programmed to update (1508), when the solution

record is found, the solution record with the price from the quote.

A further embodiment of the solutions server, shown in Figure 16, provides the processor further programmed to create a new solution record (1510), when the solution record is not found, having the same problem type, side, and symbol as the problem definition record having the same side and symbol as the quote, the same market as the quote, and the same price as the quote. A further embodiment of the solutions server, shown in Figure 16, provides the processor further programmed to delete (1512) solution records having the same side, symbol, and market as the quote, when a quote is received and the quote tag indicates that the quote is closed.

Turning now to Figure 17, additional embodiments of a solutions server are seen. One embodiment illustrated in Figure 17 provides a processor programmed to repeatedly find (1602) a next subscription record so that each existing subscription record is found in turn.

A further embodiment shown in Figure 17 provides the processor further programmed to find (1604), at least one subscription record is found, for each found subscription record, a related record of data communications parameters for the client identified in the found subscription record. A further embodiment shown in Figure 17 provides the processor further programmed to find (1606), for each found subscription record, at least one related problem definition record.

A still further embodiment shown in Figure 17 provides the processor further programmed to find (1608), when at least one problem definition record is found, for each found problem definition record, at least one related solution record. A further embodiment shown in Figure 17 provides the processor further programmed to communicate (1610), dependent upon data communication parameters identified in a found record of data communications parameters, to the client identified in the found

subscription record at least one data element of a found solution record.

Turning to Figure 18, further embodiments of the invention are seen. One embodiment illustrated in Figure 18 provides a solutions server processor (5) is programmed to communicate solutions (1704) to clients, the processor (5) programmed to communicate (1704) solutions (316) to at least one order processing system (1708) on a broker-dealer computer (1706), the broker-dealer computer (1706) further comprising a broker-dealer processor (1702).

A further embodiment shown in Figure 18 provides the broker-dealer processor (1702) programmed to receive (1710) at least one customer order (1720). In some embodiments the order processing system is the client (26) as shown on Figure 12. In embodiments of the kind shown in Figure 12, the client order processing system is installed and operated on a computer (28) separate from the computer (30) on which the solution server (5) is installed. In other embodiments, as shown in Figure 13, the client order processing system (26) is installed and operated on the same computer (30) with the solution server (5). All these embodiments, as well as other configurations of solution server and client installation and operation, are well within the scope of the present invention.

In some embodiments, the customer order (606) comprises data elements identifying symbol (210), quantity (228), and optionally, price (230), market (224), and order type (226), as shown in Figure 2H. Other embodiments use other data structures for orders, all structures for order being well within the scope of the present invention.

In many embodiments, the customer order type (226) has a relation (as shown, for example, at reference number 1728 on Figure 18) to a problem type (1712), including, for example, a one-to-one correspondence. All relations between order type and problem type are within the scope of the present invention.

A still further embodiment shown in Figure 18 provides the broker-dealer processor (1702) further programmed to find (1714) at least one solution record (1722) dependent upon the received customer order and, in many embodiments, also dependent upon having the problem type (1712).

A still further embodiment shown in Figure 18 provides the broker-dealer processor (1702) further programmed to create (1724) and send (1716) to at least one market (614) at least one solution order (1726) dependent upon the customer order (1720) and the found solution record (1722). Solution orders (1726) in many embodiments have structures dictated by the markets to which the solution orders are directed, although in many embodiments, the structure of solution orders is similar to the structure of the customer order as shown in Figure 2H. All data structures useful for solutions orders sent to markets are well within the invention. Typical embodiments of the kind illustrated in Figure 18 provide the broker-dealer processor (1702) programmed to send (1716) to at least one market (614) at least one solution order (1726), the solution order comprising the side, symbol, quantity, price, and market data elements from the found solution record.

Turning now to Figure 19, a further aspect of the invention is seen, that is, a method of providing solutions for trading securities. One embodiment illustrated at Figure 19 includes receiving a level-two quote (2002), the level-two quote comprising a symbol (2010) and at least one market participant quote (2004), the market participant quote further comprising a quote price (2006), a quote quantity (2008), a quote MPID (2012), and a quote side (2012). The illustrated embodiment includes creating (1904) in computer memory a solution set (1902) comprising at least one solution record (1906) corresponding to each market participant quote (2004) in the level-two quote (2002), the solution record (1906) comprising a solution symbol (1908), a solution side (1916), a solution MPID (1918), a solution price (1920), a solution quantity (1922), and a solution latency (1924), the solution latency comprising a latency for the market identified by the

solution MPID (1918). A further embodiment shown also in Figure 19 includes sending (1928) the solution set to at least one client (1930).

It is worth noting that embodiments of the kind illustrated in Figure 19 do not include problem definitions in the form of data structures stored in records or tables in databases. In these embodiments, problem definitions typically take, for example, forms such as "Buy MSFT" or "Sell GMC," that is, a side combined with a symbol. In embodiments using such problem definitions, comprising as they do merely a side and a symbol, the software programs stored in memory and controlling a processor can be designed to infer problem definitions with no need for separate storage of problem definitions in a database. In such embodiments, specific problem definitions are inferred from the client's presentation of the problem. For example, in the case of securities trading, a customer order includes a side and a symbol. If the order side is "bid" and the order symbol is "MSFT," then the problem definition is inferred to be "Buy MSFT." The client software in such embodiments proceeds directly to a search of the solutions records based on merely those two fields, side and symbol, to find solutions to the problem of "Buy MSFT."

In a further embodiment, as shown in Figure 21, the client (1930) comprises an automated system for trading securities, and the embodiment includes storing (2102) the sent solution set (2104) in computer memory (2106) in the client system (1930). The embodiment illustrated in Figure 21 also provides for using (2114) at least one of the sent solution records (2104) to create (2108) orders (2112) for securities. The embodiment of Figure 21 also includes deleting (2110) from computer memory the used solution records (2116).

In a further embodiment, shown in Figure 22, the client (1930) comprises more than one automated system (2204) for trading securities and the automated systems for trading securities are scaled. "Scaled" means that solutions provided to clients are apportioned

for efficiency among more than one client computer system. In the embodiment shown in Figure 22, for example, solutions for symbols beginning with the letters of the alphabet between "A" and "C" are sent to a first client system (2206), and solutions for symbols beginning with letters in other ranges are apportioned among other client systems (2208, 2210).

In a further embodiment, referring again to Figure 19, creating (1904) a solution set is seen to further comprise recording (1908) in the solution record (1906) the solution side (1916) derived from the quote side (2012), the solution MPID (1918) derived from the quote MPID (2010), the solution price (1920) derived from the quote price (2000), and the solution symbol (1908) derived from the symbol (2010) in the level-two quote (2002). In a still further embodiment, also shown in Figure 19, creating (1904) a solution set includes calculating (1910) the solution quantity (1922) for the solution record (1906) dependent upon the quote quantity (2008) and dependent upon a hidden quantity ratio (2016) for the market identified by the solution MPID (1918). This embodiment also provides for recording (1940) the solution quantity (1922) in the solution record (1906). In a further embodiment of the invention, illustrated at Figure 23, the hidden quantity ratio (2016) comprises a running average (2302) of the ratios (2304) of order fill quantity (2316) to quote quantity, the order fill quantity and the quote quantity being derived from trade data (2322) comprising descriptions of executions of orders for securities.

In a still further embodiment of the invention, illustrated at Figure 23, the hidden quantity ratio (2016) comprises a ratio (2304) of an order fill quantity to a quote quantity, the order fill quantity and the quote quantity being derived from trade data comprising descriptions of executions of orders for securities.

In a further embodiment, illustrated also in Figure 19, the solution set comprises at least two solution records, and the illustrated embodiment includes sorting (1932) the solution records (1906) to yield sorted solution records (1934). Some embodiments sort

according to side. Other embodiments sort according to price, latency, price and latency, or side and price and latency. Still other embodiments utilize other sorting principles, all sorting arrangement being well within the scope of the invention.

A further embodiment, shown also in Figure 19, provides for deleting (1936) solution latency from the sorted solution records (1934). A further embodiment, also shown on Figure 19, include creating (1950) an index (1952) for use in accessing (1954) solution records (1906) in the solution set (1902). Some embodiments index according to side. Other embodiments index according to price, latency, price and latency, or side and price and latency. All forms of index are well within the scope of the invention.
In a further embodiment of the invention, as shown in Figure 19, the solution record (1906) further comprises a type code (1926).

In a still further embodiment of the invention, as shown in Figure 20A, latency (2014) comprises the difference between the time when a broker-dealer (2018) receives (2024) from a market (2020) a response to an order and the time when the order was sent (2022) to the market. Other embodiments measure latency according to the time for the order to travel from the broker-dealer to the market. Other embodiments measure latency according to the fill time within the market. All measures of latency as used in various embodiments are well within the scope of the invention.

Turning now to Figure 24, a further aspect of the invention is seen, that is, a system for generating solutions for trading securities. In one embodiment, shown in Figure 24, a system (2402) for generating solutions for trading securities includes means for receiving (2402) a level-two quote (2002), the level-two quote comprising a symbol and at least one market participant quote (2005), the market participant quote further comprising a quote price (2006), a quote quantity (2008), a quote MPID (2010), and a quote side (2012). Means for receiving a level-two quote includes Nasdaq feeds and subscriber feeds from ECNs received across networks or dedicated lines through communications

ports operated under program control in a computer system. Other sources of feeds for level-two quotes are within the scope of the invention, including dedicated communications hardware which in some cases is supplied by the sources of quotes for the purpose of communicating quotes, including, for example, Nasdaq's "Service Delivery Platform" or "SDP."

The embodiment illustrated in Figure 24 includes also means for creating (2406) in computer memory (2408) a solution set (1902) comprising at least one solution record (1906) corresponding to each market participant quote (2005) in the level-two quote (2002), the solution record (1906) comprising a solution symbol (1908), a solution side (1916), a solution MPID (1918), a solution price (1920), a solution quantity (1922), and a solution latency (1924), the solution latency comprising a latency for a market identified by the solution MPID (1918). Means for creating a solution set in computer memory, in most embodiments, is a computer processor coupled to computer memory and operating under control of a program stored in computer memory. Forms of computer memory operable within the invention include random access memory, read-only memory, programmable read-only memory, erasable programmable read-only memory, other forms of semiconductor memory, as well as various forms of magnetic storage such as computer disk drives.

The embodiment illustrated in Figure 24 also includes means for sending (2410) the solution set to at least one client (1930). Means for sending the solution in such embodiments includes data communications ports, networks, satellite links, dedicated phone lines, intranets, internets, extranets, and other forms of networks, coupling the embodiment of the invention to at least one client.

In a further embodiment, shown in Figure 25, the client (1930) comprises an automated system for trading securities that includes means for storing (2502) the sent solution set (1902) in computer memory (2504) in the client (1930). Means for storing a solution set

in computer memory, in most embodiments, is a computer processor coupled to computer memory and operating under control of a program stored in computer memory. Forms of computer memory operable within the invention include random access memory, read-only memory, programmable read-only memory, erasable programmable read-only memory, other forms of semiconductor memory, as well as various forms of magnetic storage such as computer disk drives.

The embodiment illustrated in Figure 25 includes also means for creating (2506), from at least one of the sent solution records (1902), an order (2508) for securities, wherein creating an order from the sent solution record (1902) further comprises creating (2512) a used solution record (2510). Means for creating an order, in most embodiments, is a computer processor coupled to computer memory and operating under control of a program stored in computer memory.

The embodiment illustrated in Figure 25 also provides means for deleting (2514) from computer memory (2504) the used solution record (2510). Means for deleting the used record, in most embodiments, is a computer processor coupled to computer memory and operating under control of a program stored in computer memory. Forms of computer memory operable within the invention include random access memory, read-only memory, programmable read-only memory, erasable programmable read-only memory, other forms of semiconductor memory, as well as various forms of magnetic storage such as computer disk drives.

In a further embodiment, as shown in Figure 22, the client (1930) includes more than one automated system (2204) for trading securities and the automated systems (2204) for trading securities are scaled. "Scaled" means that solutions provided to clients are apportioned for efficiency among more than one client computer system. In the embodiment shown in Figure 22, for example, solutions for symbols beginning with the

letters of the alphabet between "A" and "C" are sent to a first client system (2206), and solutions for symbols beginning with letters in other ranges are apportioned among other client systems (2208, 2210).

In a further embodiment, shown in Figure 24, means for creating a solution set includes means for recording (2412) in the solution record, as shown in more detail in Figure 29, the solution side (1916) derived from the quote side (2012), the solution MPID (1918) derived from the quote MPID (2010), the solution price (1920) derived from the quote price (2006), and the solution symbol (1908) derived from the symbol (2004) in the level-two quote. Means for recording in the solution record, in most embodiments, is a computer processor coupled to computer memory and operating under control of a program stored in computer memory.

In a further embodiment of the invention illustrated in Figure 24 the means for creating a solution set includes means for calculating (2414) the solution quantity for the solution record dependent upon the quote quantity and dependent upon a hidden quantity ratio for the market identified by the solution MPID. Means for calculating the solution quantity, in most embodiments, is a computer processor coupled to computer memory and operating under control of a program stored in computer memory.

In the embodiment illustrated in Figure 24, the means for creating a solution set includes means for recording (2416) the solution quantity in the solution record. Means for calculating the solution quantity and means for recording the solution quantity, in most embodiments, are a computer processor coupled to computer memory and operating under control of a program stored in computer memory.

In a further embodiment of the invention, as shown in Figure 23, the hidden quantity ratio (2016) comprises a running average (2302) of the ratios (2304) of order fill quantity (2316) to quote quantity (2318), the order fill quantity and the quote quantity being

derived from trade data (2322) comprising descriptions of executions of orders for securities. In a still further embodiment of the invention, also shown in Figure 23, the hidden quantity ratio (2016) comprises a ratio (2304) of an order fill quantity (2316) to a quote quantity (2318), the order fill quantity and the quote quantity being derived from trade data (2322) comprising descriptions of executions of orders for securities.

In a further embodiment of the invention, as shown in Figure 24, the solution set (1902) includes at least two solution records (1906), and the embodiment further includes means for sorting (2418) the solution records to yield sorted solution records (1934). A further embodiment, as shown in Figure 24, includes means for sorting the solution records according to side. Other embodiments includes means for sorting the solution records according to price, latency, price and latency, side and price and latency. Other embodiments sort according to other fields or combinations of fields within the solution records. All sorting arrangements of the solution records, in various alternative embodiments, are well within the scope of the invention. Means for sorting solution records, in most embodiments, is a computer processor coupled to computer memory and operating under control of a program stored in computer memory.

A further embodiment, also shown in Figure 24, provides means for deleting (2420) solution latency from the sorted solution records (1934). Means for deleting in most embodiments is at least one computer processor operating under control of a program stored in computer memory.

A further embodiment, also shown in Figure 24, includes means for creating (2422) an index for the solution set. Means for creating an index in most embodiments is at least one computer processor operating under program control to read index fields from the solution records and create a new set of ordered index records in computer memory dependent upon the read fields. Indexes so created are ordered according to side, price,

latency, price and latency, side and price and latency, and other ordering principles, all of which in various alternative embodiments are well within the scope of the invention.

In a further embodiment of the invention, shown in Figure 24, the solution record (1906) includes a type code (1926). In many embodiments, the type code (1926) is used to select among computer program subroutines to vary the function of the invention to provide solutions optimizing speed of order execution, solutions optimizing price, solution optimizing quantities of securities traded, and quantities optimizing other parameters of performance. Many such solutions are effected by use of various sorting principles applied to solution records in computer memory.

In a further embodiment of the invention, shown in Figure 20A, solution latency (1924) comprises the difference between the time (2018) when a broker-dealer (2024) receives from a market (2020) a response to an order and the time when the order was sent (2022) to the market (2020). Other embodiments measure latency according to the time for the order to travel from the broker-dealer to the market. Other embodiments measure latency according to the fill time within the market. All measures of latency as used in various embodiments are well within the scope of the invention.

Turning now to Figure 26, a further aspect of the invention is seen as a system (2602) for providing solutions for trading securities. One embodiment, illustrated in Figure 26, provides a processor (2604) coupled (2610) to at least one source of quotes (2606) and coupled (2612) to at least one client (1930), the processor (2604) programmed to receive (2608) at least one level-two quote (2002), the level-two quote (2002) comprising a symbol and at least one market participant quote (2004), the market participant quote comprising a quote price (2006), a quote quantity (2008), a quote MPID (2010), and a quote side (2012).

A further embodiment illustrated in Figure 26 includes the processor programmed to create (2614) a solution set (1902) comprising at least one solution record (1906) corresponding to each market participant quote (2004), the solution record comprising a solution symbol (1908), a solution side (1906), a solution MPID (1918), a solution price (1920), a solution quantity (1922), and a solution latency (1924), the solution latency comprising a latency for the market identified by the solution MPID (1918). The embodiment illustrated in Figure 26 includes also a memory (2618) coupled (2620) to the processor (2604) with the processor programmed to store (2616) the solution set (1902) in the memory (2618).

A further embodiment illustrated in Figure 26 includes the processor the processor (2604) is programmed to send (2622) the solution set (1902) to the client (1930).

In a further embodiment, shown in Figure 27, the client (1930) comprises an automated system for trading securities, the system further comprising a client processor (2702) coupled to client memory (2704). In the embodiment shown in Figure 27, the client processor is programmed to store (2706) the sent solution set (1902) in client memory (2704), create (2708), dependent upon the sent solution set (1902), orders (2710) for securities, and send (2712) the orders for securities to markets (2714).

In a further embodiment, shown in Figure 28, the client (1930) comprises more than one automated system (2804) for trading securities, and the automated systems for trading securities are scaled (2806, 2808, 2810). "Scaled" means that solutions provided to clients are apportioned for efficiency among more than one client computer system. In the embodiment shown in Figure 28, for example, solutions for symbols beginning with the letters of the alphabet between "A" and "C" are sent to a first client system (2806), and solutions for symbols beginning with letters in other ranges are apportioned among other client systems (2808, 2810).

In a further embodiment, shown in Figure 26, the computer processor is programmed also to record (2624) in the solution set (1902), as shown in more detail in Figure 29, the solution side (1916) derived from the quote side (2012), the solution MPID (1918) derived from the quote MPID (2010), the solution price (1920) derived from the quote price (2006), and the solution symbol (1908) derived from the symbol (2004) in the level-two quote. In a further embodiment, as shown in Figure 26, the processor is further programmed to calculate (2626) the solution quantity dependent upon the quote and dependent upon a hidden quantity ratio (2628). The processor in this embodiment is programmed also to record (2624) the solution quantity in the solution record (2628).

In a further embodiment, illustrated in Figure 23, the hidden quantity ratio (2016) comprises a running average (2302) of the ratios (2304) of order fill quantity (2316) to quote quantity (2318), the order fill quantity and the quote quantity being derived from trade data (2322) comprising descriptions of executions of orders for securities having structure shown at 2322 in Fig. 23. In a still further embodiment, also illustrated in Figure 23, the hidden quantity ratio (2016) comprises a ratio (2304) of an order fill quantity (2316) to a quote quantity (2318), the order fill quantity and the quote quantity being derived from trade data (2322) comprising descriptions of executions of orders for securities.

In a further embodiment, shown in Figure 26, the solution set includes at least two solution (2628) records, the processor (2604) being further programmed to sort (2630) the solution records (2628) to yield a sorted solution set (2632). In a further embodiment, also shown in Figure 26, the processor is further programmed to sort the solution records according to side. In other embodiments, the processor is programmed to sort the solution records according to price, latency, price and latency, or side and price and latency. Other embodiments use other sorting arrangements, all of which are well within the scope of the invention.

In a further embodiment, shown in Figure 26, the processor is further programmed to delete (2636) solution latency from the sorted solution records (2632). In a further embodiment, illustrated in Figure 26A, the processor (2604) is further programmed to create (2634) an index (2634) from the solution records (2628) in the solution set (1902) for use in improved access (2642) to the solution records (2628). Indexes so created are ordered, in various alternative embodiments, according to side, price, latency, price and latency, side and price and latency, and other ordering principles. These, as well as all other arrangements of indexes for improving access to the solution records, in various alternative embodiments, are well within the scope of the invention.

In a further embodiment, illustrated in Figure 26, the solution record (1906) further includes a type code (1926). In many embodiments, the type code (1926) is used to select among computer program subroutines to vary the function of the invention to provide solutions optimizing speed of order execution, solutions optimizing price, solution optimizing quantities of securities traded, and quantities optimizing other parameters of performance. Many such solutions are effected by use of various sorting principles applied to solution records in computer memory.

In a further embodiment, shown in Figure 20A, latency comprises the difference between the time when a broker-dealer receives from a market a response to an order and the time when the order was sent to the market. Other embodiments measure latency according to the time for the order to travel from the broker-dealer to the market. Other embodiments measure latency according to the fill time within the market. All measures of latency as used in various embodiments are well within the scope of the invention.

An example of operation of the invention as a system for trading securities is the following. Assume that a reference table contains the following information regarding latency and hidden quantity ratios of markets.

Reference Table		
MPID	**LATENCY**	**HQR**
MSCO	SN	0.50
ISLD	52	1.50
INCA	475	1.25
GSCO	SN	0.75
MADF	SN	0.75
ARCA	385	1
SN	3500	NA

In most embodiments, MPIDs in the table will represent all currently-active markets. Note that all market makers will receive any orders sent them either through SelectNet or SOES and therefore must use the execution speed of those systems.

Upon receiving the following level-two quote the system will create the following solution record. Only the bid side is illustrated. The solution record as illustrated is already sorted according to price.

Level-Two Quote (Bid)		
Bid MPID	**Bid Price**	**Bid Quantity**
MSCO	99	1
GSCO	99	2
ISLD	99	1
MADF	98	10
ARCA	98	8
INCA	98	7

Sorting according to price and latency produces the following form of solution.

Solution (Bid Side Only)		
MPID	**Price**	**Quantity**
ISLD	99	1
MSCO	99	1
GSCO	99	2
ARCA	98	8
INCA	98	7
MADF	98	10

Calculating a calculated quantity dependent upon bid quantity and hidden quantity ratio is illustrated in the following table:

Bid MPID	Bid Price	Bid Quantity	***HQR***	Calculated Quantity	Market Adjusted Quantity
ISLD	99	1	1.50	1.5	1.5
MSCO	99	1	0.50	0.5	1
GSCO	99	2	0.75	1.5	1.5
ARCA	98	8	1	8	8
INCA	98	7	1.25	8.75	8.75
MADF	98	10	0.75	7.5	7.5

Market Adjusted Quantity is the result of applying market rules to the calculated quantity. For example, orders through SelectNet must be for at least 100 shares, and some markets have rules for preferencing market participants who are not currently priced at the inside price. Note that in this example embodiment, calculating the calculated quantity can be done before or after sorting the solution records.

In many embodiments, determination will be made for each market whether to use the calculated quantity or the quoted quantity. Because ECN quotes are usually representations of actual orders, ECN quotes are relatively firm. However, many ECNs support forms of hidden orders. Even for ECN's, therefore, there is often opportunity to execute quantities larger than their respective quoted quantities. In this example, it is

determined that the opportunity to execute against hidden orders in ISLD,ARCA and INCA is significant and the calculated quantity is used, resulting in the following solution, dependent upon the liquidity (quoted quantity) of the stock.

Solution (Bid Side Only)		
MPID	**Price**	**Quantity**
ISLD	99	1.5
GSCO	99	2
MSCO	99	1
INCA	98	8.75
ARCA	98	8
MADF	98	10
BBO	99	1.5

This solution in this example is provided to at least one client. If the client is an automated system for trading securities, the example processing continues as follows. Assume that the offer side of the current pertinent level-two quote is the following:

Level-Two Quote (for a security represented by a symbol)	
Quantity	**Price**
1000	98 ½
5000	99

On that assumption, use of the new solution in the embodiment under discussion would cause the following orders to be generated in response to a customer order to sell 1000 shares, trading 1000 shares at 98 ½ as follows:

Orders					
MPID	Side	Quantity	Price	Via	Price Improve-

					ment
ISLD	S	150	99		½
GSCO	S	200	99	SOES	½
MSCO	S	100	99	SOES	½
INCA	S	550	99		½

After these trades are executed, the remaining offer side of the current level-two quote is 5000 shares at $99.00. The remaining unused records on the bid side of the solution set would be:

Solution (Bid Side Only) (remaining unused solution records)		
MPID	**Price**	**Quantity**
INCA	98	3.75
ARCA	98	8
MADF	98	10

The "used" solution records are removed in this example embodiment to avoid sending new orders to markets that may no longer have sufficient quantity to fill orders. Sending orders to markets with increased risk of failure to fill is potentially costly in terms of overall execution time for customers' orders. In this example it is useful to note that the used solution record for the Island ECN (MPID = ISLD) was deleted from the set of solution records remaining after the trade. The ISLD solution record was deleted despite the fact that ISLD shows an hidden quantity ratio of 1.5, indicating that ISLD may fill orders for quantities substantially larger than ISLD's quoted quantities for a security, therefore identifying ISLD as a natural trader, in this case a natural seller, of the subject securities. Nevertheless, many such embodiments will delete the ISLD solution record after using it to develop an order because it may not be possible to know or infer whether ISLD will continue to sell securities at the previous quoted price. Continuing to order at that price might risk a time-consuming order round trip, rejection for nothing. In such embodiments, the system waits for ISLD to refresh its quote and uses the new quote price

with the quote quantity and the hidden quantity ratio to generate a new solution record for ISLD. This approach addresses the problem of hidden liquidity by use of the hidden quantity ratio on a new quote, rather than by leaving the ISLD solution record in the solution set in reliance on the previous quote.

Turning now to Figure 30, an additional aspect of the invention is seen, that is, a method (3002) of creating an improved level-two quote. One embodiment shown in Figure 30 includes receiving (3006) in an automated system (3004), which system comprises at least one computer processor (3008) coupled to computer memory (3010), a level-two quote (3012), the level-two quote comprising a data format further comprising a symbol (3014) and at least one market participant quote (3016), the market participant quote further comprising a quote price (3018), a quote quantity (3020), a quote MPID (3022), and a quote side (3024).

The first embodiment illustrated in Figure 30 also includes creating (3026) in computer memory (3010) an improved level-two quote (3028) comprising a symbol (3014) and at least one improved market participant quote (3030), which improved market participant quote comprises a side (3032), an MPID (3034), a price (3036), a quantity (3038) dependent upon the market participant quote quantity (3020) and also dependent upon a hidden quantity ratio (3042). An additional embodiment shown on Figure 30 provides in the improved market participant quote a latency (3046), the latency comprising a latency for the market identified by the MPID (3034).

A further embodiment shown in Figure 30 includes the improved level-two quote (3028) comprising at least two improved market participant quotes (3030) wherein the improved market participant quotes are sorted (3044). Some embodiments sort according to side. Other embodiments sort according to price, latency, price and latency, or side and price and latency. Still other embodiments utilize other sorting principles, all sorting

arrangement being well within the scope of the invention.

The illustrated embodiments of Figure 30 include providing (3052) the improved level-two quotes (3028) to clients (3050), including providing the improved level-two quote to clients in the form of streaming serial data provided to clients by use of suitable means for data communications (3048). Suitable means for data communications, useful in various embodiments, includes networks, dedicated satellite channels, dedicated telephone lines, and the like. Any form of data communications adapted to stream data in the form of level-two quotes is well within the scope of the invention. In addition to other data elements comprising an improved level-two quote, some embodiments make available to clients for display, or for other uses, at least one indication of hidden liquidity, such as, for example, a hidden quantity ratio. The improved level-two quotes are provided to clients for use in trading, investment decision-making, or for display at the client's option. Clients for the invention include market participants, electronic market participants, investors, traders, institutions, market makers, ECNs, websites, web pages, broadcast or cable television channels, and any other clients interested in streaming market data.